Kernel Application (KERNEL)

version 2.11
Typeset in \LaTeX\ from SGML source using the DOCBUILDER 3.3.2 Document System.
## Contents

**1 Kernel Reference Manual**

1.1 kernel ............................................. 30  
1.2 application ........................................ 34  
1.3 auth ............................................. 43  
1.4 code ............................................. 45  
1.5 disk_log ........................................ 53  
1.6 erl_boot_server ................................. 67  
1.7 erl_ddll ......................................... 69  
1.8 erl_prim_loader .................................. 71  
1.9 erlang ............................................ 74  
1.10 error_handler ................................... 138  
1.11 error_logger ..................................... 140  
1.12 file .............................................. 146  
1.13 gen_tcp ......................................... 168  
1.14 gen_udp ......................................... 173  
1.15 global ............................................ 176  
1.16 global_group .................................... 181  
1.17 heart ............................................. 185  
1.18 inet .............................................. 187  
1.19 init .............................................. 196  
1.20 netadm ........................................... 201  
1.21 net_kernel ...................................... 204  
1.22 os ............................................... 208  
1.23 packages ......................................... 211  
1.24 pg2 .............................................. 214  
1.25 rpc .............................................. 216  
1.26 seq_trace ....................................... 223  
1.27 user .............................................. 231  
1.28 wrap_log_reader ............................... 232  
1.29 zlib ............................................. 234

Kernel Application (KERNEL)
Kernel Reference Manual

Short Summaries

- Application kernel [page 30] - The Kernel Application
- Erlang Module application [page 34] - Generic OTP application functions
- Erlang Module auth [page 43] - Erlang Network Authentication Server
- Erlang Module code [page 45] - Erlang Code Server
- Erlang Module disk_log [page 53] - A disk based term logging facility
- Erlang Module erl_boot_server [page 67] - Boot Server for Other Erlang Machines
- Erlang Module erl_ddll [page 69] - Dynamic Driver Loader and Linker
- Erlang Module erl_prim_loader [page 71] - Low Level Erlang Loader
- Erlang Module erlang [page 74] - The Erlang BIFs
- Erlang Module error_handler [page 138] - Default System Error Handler
- Erlang Module error_logger [page 140] - Erlang Error Logger
- Erlang Module file [page 146] - File Interface Module
- Erlang Module gen_tcp [page 168] - Interface to TCP/IP sockets
- Erlang Module gen_udp [page 173] - Interface to UDP sockets
- Erlang Module global [page 176] - A Global Name Registration Facility
- Erlang Module global_group [page 181] - Grouping Nodes to Global Name Registration Groups
- Erlang Module heart [page 185] - Heartbeat Monitoring of an Erlang Runtime System
- Erlang Module inet [page 187] - Access to TCP/IP Protocols
- Erlang Module init [page 196] - Coordination of System Startup
- Erlang Module net_adm [page 201] - Various Erlang Net Administration Routines
- Erlang Module net_kernel [page 204] - Erlang Networking Kernel
- Erlang Module os [page 208] - Operating System Specific Functions
- Erlang Module packages [page 211] - Packages in Erlang
- Erlang Module pg2 [page 214] - Distributed Named Process Groups
- Erlang Module rpc [page 216] - Remote Procedure Call Services
- Erlang Module seq_trace [page 223] - Sequential Tracing of Messages
- Erlang Module user [page 231] - Standard I/O Server
- Erlang Module wrap_log_reader [page 232] - A function to read internally formatted wrap disk logs

kernel

No functions are exported.

application

The following functions are exported:

- `get_all_env()` -> `Env`  
  [page 34] Get the configuration parameters for an application

- `get_all_env(Application)` -> `Env`  
  [page 34] Get the configuration parameters for an application

- `get_all_key()` -> `ok, Keys` | []  
  [page 34] Get the application specification keys

- `get_all_key(Application)` -> `ok, Keys` | undefined  
  [page 34] Get the application specification keys

- `get_application()` -> `ok, Application` | undefined  
  [page 35] Get the name of an application containing a certain process or module

- `get_application(Pid | Module)` -> `ok, Application` | undefined  
  [page 35] Get the name of an application containing a certain process or module

- `get_env(Par)` -> `ok, Val` | undefined  
  [page 35] Get the value of a configuration parameter

- `get_env(Application, Par)` -> `ok, Val` | undefined  
  [page 35] Get the value of a configuration parameter

- `get_key(Key)` -> `ok, Val` | undefined  
  [page 35] Get the value of an application specification key

- `get_key(Application, Key)` -> `ok, Val` | undefined  
  [page 35] Get the value of an application specification key

- `load(AppDescr)` -> `ok` | {error, Reason}  
  [page 35] Load an application

- `load(AppDescr, Distributed)` -> `ok` | {error, Reason}  
  [page 35] Load an application

- `loaded_applications()` -> `[(Application, Description, Vsn)]`  
  [page 36] Get the currently loaded applications

- `permit(Application, Bool)` -> `ok` | {error, Reason}  
  [page 36] Change an application's permission to run on a node.

- `set_env(Application, Par, Val)` -> `ok`  
  [page 37] Set the value of a configuration parameter

- `set_env(Application, Par, Val, Timeout)` -> `ok`  
  [page 37] Set the value of a configuration parameter

- `start(Application)` -> `ok` | {error, Reason}  
  [page 37] Load and start an application
The following functions are exported:

- `is_auth(Node) -> yes | no`
  [page 43] Status of communication authorization (deprecated)

- `cookie() -> Cookie`
  [page 43] Magic cookie for local node (deprecated)

- `cookie(TheCookie) -> true`
  [page 43] Set the magic for the local node (deprecated)

- `node_cookie([Node, Cookie]) -> yes | no`
  [page 43] Set the magic cookie for a node and verify authorization (deprecated)

- `node_cookie(Node, Cookie) -> yes | no`
  [page 43] Set the magic cookie for a node and verify authorization (deprecated)
The following functions are exported:

- `set_path(Path)` -> true | {error, What}
  [page 46] Set the code server search path
- `get_path()` -> Path
  [page 47] Return the code server search path
- `add_path(Dir)` -> true | {error, What}
  [page 47] Add a directory to the end of the code path
- `add_pathz(Dir)` -> true | {error, What}
  [page 47] Add a directory to the end of the code path
- `add_patha(Dir)` -> true | {error, What}
  [page 47] Add a directory to the beginning of the code path
- `add_paths(Dirs)` -> ok
  [page 47] Add directories to the end of the code path
- `add_pathsz(Dirs)` -> ok
  [page 47] Add directories to the end of the code path
- `add_pathsa(Dirs)` -> ok
  [page 47] Add directories to the beginning of the code path
- `del_path(Name | Dir)` -> true | false | {error, What}
  [page 48] Delete a directory from the code path
- `replace_path(Name, Dir)` -> true | {error, What}
  [page 48] Replace a directory with another in the code path
- `load_file(Module)` -> {module, Module} | {error, What}
  [page 48] Load a module
- `load_abs(Filename)` -> {module, Module} | {error, What}
  [page 48] Load a module, residing in a given file
- `ensure_loaded(Module)` -> {module, Module} | {error, What}
  [page 49] Ensure that a module is loaded
- `load_binary(Module, Filename, Binary)` -> {module, Module} | {error, What}
  [page 49] Load object code for a module
- `delete(Module)` -> true | false
  [page 49] Removes current code for a module
- `purge(Module)` -> true | false
  [page 49] Removes old code for a module
- `soft_purge(Module)` -> true | false
  [page 50] Removes old code for a module, unless no process uses it
- `is_loaded(Module)` -> {file, Loaded} | false
  [page 50] Check if a module is loaded
- `all_loaded()` -> [{Module, Loaded}]
  [page 50] Get all loaded modules
- `which(Module)` -> Which
  [page 50] The object code file of a module
- `get_object_code(Module)` -> {Module, Binary, Filename} | error
  [page 50] Get the object code for a module
Kernel Reference Manual

- **root_dir()** -> RootDir
  [page 51] Root directory of Erlang/OTP

- **lib_dir()** -> LibDir
  [page 51] Library directory of Erlang/OTP

- **lib_dir(Name)** -> Dir | {error, What}
  [page 51] Library directory for an application

- **compiler_dir()** -> Dir
  [page 51] Library directory for the compiler

- **priv_dir(Name)** -> PrivDir | {error, What}
  [page 51] Priv directory for an application

- **objfile_extension()** -> Ext
  [page 52] Object code file extension

- **stick_dir(Dir)** -> ok | {error, What}
  [page 52] Mark a directory as sticky

- **unstick_dir(Dir)** -> ok | {error, What}
  [page 52] Remove a sticky directory mark

- **rehash()** -> ok
  [page 52] Rehash or create code path cache

- **where_is_file(Filename)** -> Absname | non_existing
  [page 52] Full name of a file located in the code path

- **clash()** -> ok
  [page 52] Search for modules with identical names.

### disk_log

The following functions are exported:

- **accessible_logs()** -> [{[LocalLog], [DistributedLog]}]
  [page 55] Return the accessible disk logs on the current node.

- **alog(Log, Term)**
  [page 55] Asynchronously log an item onto a disk log.

- **balog(Log, Bytes)** -> ok | {error, Reason}
  [page 55] Asynchronously log an item onto a disk log.

- **alog_terms(Log, TermList)**
  [page 55] Asynchronously log several items onto a disk log.

- **balog_terms(Log, BytesList)** -> ok | {error, Reason}
  [page 55] Asynchronously log several items onto a disk log.

- **block(Log)**
  [page 56] Block a disk log.

- **block(Log, QueueLogRecords)** -> ok | {error, Reason}
  [page 56] Block a disk log.

- **change_header(Log, Header)** -> ok | {error, Reason}
  [page 56] Change the head or head_func option for an owner of a disk log.

- **change_notify(Log, Owner, Notify)** -> ok | {error, Reason}
  [page 56] Change the notify option for an owner of a disk log.

- **change_size(Log, Size)** -> ok | {error, Reason}
  [page 56] Change the size of an open disk log.
- chunk(Log, Continuation)
  [page 57] Read a chunk of objects written to a disk log.
- chunk(Log, Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | eof | {error, Reason}
  [page 57] Read a chunk of objects written to a disk log.
- bchunk(Log, Continuation)
  [page 57] Read a chunk of objects written to a disk log.
- bchunk(Log, Continuation, N) -> {Continuation2, Binaries} | {Continuation2, Binaries, Badbytes} | eof | {error, Reason}
  [page 57] Read a chunk of objects written to a disk log.
- chunk_info(Continuation) -> InfoList | {error, Reason}
  [page 58] Return information about a chunk continuation of a disk log.
- chunk_step(Log, Continuation, Step) -> {ok, Continuation2} | {error, Reason}
  [page 58] Step forward or backward among the wrap log files of a disk log.
- close(Log) -> ok | {error, Reason}
  [page 59] Close a disk log.
- format_error(Error) -> Chars
  [page 59] Return an English description of a disk log error reply.
- inc_wrap_file(Log) -> ok | {error, Reason}
  [page 59] Change to the next wrap log file of a disk log.
- info(Log) -> InfoList | {error, no_such_log}
  [page 59] Return information about a disk log.
- 1close(Log)
  [page 61] Close a disk log on one node.
- 1close(Log, Node) -> ok | {error, Reason}
  [page 61] Close a disk log on one node.
- log(Log, Term)
  [page 61] Log an item onto a disk log.
- blog(Log, Bytes) -> ok | {error, Reason}
  [page 61] Log an item onto a disk log.
- log_terms(Log, TermList)
  [page 61] Log several items onto a disk log.
- blog_terms(Log, BytesList) -> ok | {error, Reason}
  [page 61] Log several items onto a disk log.
- open(ArgL) -> OpenRet | DistOpenRet
- pid2name(Pid) -> {ok, Log} | undefined
  [page 65] Return the name of the disk log handled by a pid.
- reopen(Log, File)
  [page 65] Reopen a disk log and save the old log.
- reopen(Log, File, Head)
  [page 65] Reopen a disk log and save the old log.
- breopen(Log, File, BHead) -> ok | {error, Reason}
  [page 65] Reopen a disk log and save the old log.
- sync(Log) -> ok | {error, Reason}
  [page 65] Flush the contents of a disk log to the disk.
Kernel Reference Manual

- `truncate(Log)`
  [page 66] Truncate a disk log.
- `truncate(Log, Head)`
  [page 66] Truncate a disk log.
- `btruncate(Log, BHead)` → ok | {error, Reason}
  [page 66] Truncate a disk log.
- `unblock(Log)` → ok | {error, Reason}
  [page 66] Unblock a disk log.

erl_boot_server

The following functions are exported:

- `start(Slaves)` → {ok, Pid} | {error, What}
  [page 67] Start the boot server
- `start_link(Slaves)` → {ok, Pid} | {error, What}
  [page 67] Start the boot server and links the caller
- `add_slave(Slave)` → ok | {error, What}
  [page 67] Add a slave to the list of allowed slaves
- `delete_slave(Slave)` → ok | {error, What}
  [page 67] Delete a slave from the list of allowed slaves
- `which_slaves()` → Slaves
  [page 68] Return the current list of allowed slave hosts

erl_ddll

The following functions are exported:

- `load_driver(Path, Name)` → {ok, ErrorDesc}
  [page 69] Load a driver
- `unload_driver(Name)` → {ok, ErrorDesc}
  [page 69] Load a driver
- `loaded_drivers()` → {ok, Drivers}
  [page 69] List loaded drivers
- `format_error(ErrorDesc)` → string()
  [page 69] Format an error descriptor

erl_prim_loader

The following functions are exported:

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

The following functions are exported:

- `abs(Number) -> int() | float()`  
  [page 74] Arithmetical absolute value

- `erlang:append_element(Tuple1, Term) -> Tuple2`  
  [page 74] Append an extra element to a tuple

- `apply(Fun, Args) -> term() | empty()`  
  [page 75] Apply a function to an argument list

- `apply(Module, Function, Args) -> term() | empty()`  
  [page 75] Apply a function to an argument list

- `atom_to_list(Atom) -> string()`  
  [page 76] Text representation of an atom

- `binary_to_list(Binary) -> [char()]`  
  [page 76] Convert a binary to a list

- `binary_to_list(Binary, Start, Stop) -> [char()]`  
  [page 76] Convert part of a binary to a list

- `binary_to_term(Binary) -> term()`  
  [page 76] Decode an Erlang external term format binary

- `erlang:bump_reductions(Reductions) -> void()`  
  [page 76] Increment the reduction counter

- `erlang:cancel_timer(TimerRef) -> Time | false`  
  [page 76] Cancel a timer

- `check_process_code(Pid, Module) -> bool()`  
  [page 77] Check if a process is executing old code for a module

- `concat_binary(ListOfBinaries)`  
  [page 77] Concatenate a list of binaries (deprecated)

- `date() -> {Year, Month, Day}`  
  [page 77] Current date

- `delete_module(Module) -> true | undefined`  
  [page 77] Make the current code for a module old

- `erlang:demonitor(MonitorRef) -> true`  
  [page 78] Stop monitoring

- `erlang:demonitor(MonitorRef, OptionList) -> true`  
  [page 78] Stop monitoring

- `disconnect_node(Node) -> bool() | ignored`  
  [page 79] Force the disconnection of a node

- `erlang:display(Term) -> true`  
  [page 79] Print a term on standard output

- `element(N, Tuple) -> term()`  
  [page 79] Get Nth element of a tuple

- `erase() -> [[]]`  
  [page 80] Return and delete the process dictionary

- `erase(Key) -> Val | undefined`  
  [page 80] Return and delete a value from the process dictionary
- `erlang:error(Reason)`
  [page 80] Stop execution with a given reason
- `erlang:error(Reason, Args)`
  [page 80] Stop execution with a given reason
- `exit(Reason)`
  [page 81] Stop execution with a given reason
- `exit(Pid, Reason) -> true`
  [page 81] Send an exit signal to a process
- `erlang:fault(Reason)`
  [page 81] Stop execution with a given reason
- `erlang:fault(Reason, Args)`
  [page 81] Stop execution with a given reason
- `float(Number) -> float()`
  [page 82] Convert a number to a float
- `float_to_list(Float) -> string()`
  [page 82] Text representation of a float
- `erlang:fun_info(Fun) -> [{Item, Info}]`
  [page 82] Information about a fun
- `erlang:fun_info(Fun, Item) -> {Item, Info}`
  [page 83] Information about a fun
- `erlang:fun_to_list(Fun) -> string()`
  [page 83] Text representation of a fun
- `erlang:function_exported(Module, Function, Arity) -> bool()`
  [page 84] Check if a function is exported and loaded
- `garbage_collect() -> true`
  [page 84] Force an immediate garbage collection of the calling process
- `garbage_collect(Pid) -> bool()`
  [page 84] Force an immediate garbage collection of a process
- `get() -> [{Key, Val}]`
  [page 84] Return the process dictionary
- `get(Key) -> Val | undefined`
  [page 84] Return a value from the process dictionary
- `erlang:get_cookie() -> Cookie | nocookie`
  [page 85] Get the magic cookie of the local node
- `get_keys(Val) -> [Key]`
  [page 85] Return a list of keys from the process dictionary
- `erlang:get_stacktrace() -> [{Module, Function, Arity | Args}]`
  [page 85] Get the call stack backtrace of the last exception
- `group_leader() -> GroupLeader`
  [page 85] Get the group leader for the calling process
- `group_leader(GroupLeader, Pid) -> true`
  [page 86] Set the group leader for a process
- `halt()`
  [page 86] Halt the Erlang runtime system and indicate normal exit to the calling environment
- `halt(Status)`
  [page 86] Halt the Erlang runtime system
- `erlang:hash(Term, Range) -> Hash`  
  [page 86] Hash function (deprecated)
- `hd(List) -> term()`  
  [page 86] Head of a list
- `erlang:hibernate(Module, Function, Args)`  
  [page 87] Hibernate a process until a message is sent to it
- `erlang:info(Type) -> Res`  
  [page 87] Information about the system (deprecated)
- `integer_to_list(Integer) -> string()`  
  [page 87] Text representation of an integer
- `erlang:integer_to_list(Integer, Base) -> string()`  
  [page 88] Text representation of an integer
- `iolist_to_binary(IoListOrBinary) -> binary()`  
  [page 88] Convert an iolist to a binary
- `iolist_size(Item) -> int()`  
  [page 88] Size of an iolist
- `is_alive() -> bool()`  
  [page 88] Check whether the local node is alive
- `is_atom(Term) -> bool()`  
  [page 88] Check whether a term is an atom
- `is_binary(Term) -> bool()`  
  [page 88] Check whether a term is a binary
- `is_boolean(Term) -> bool()`  
  [page 89] Check whether a term is a boolean
- `erlang:is_builtin(Module, Function, Arity) -> bool()`  
  [page 89] Check if a function is a BIF implemented in C
- `is_float(Term) -> bool()`  
  [page 89] Check whether a term is a float
- `is_function(Term) -> bool()`  
  [page 89] Check whether a term is a fun
- `is_function(Term, Arity) -> bool()`  
  [page 89] Check whether a term is a fun with a given arity
- `is_integer(Term) -> bool()`  
  [page 90] Check whether a term is an integer
- `is_list(Term) -> bool()`  
  [page 90] Check whether a term is a list
- `is_number(Term) -> bool()`  
  [page 90] Check whether a term is a number
- `is_pid(Term) -> bool()`  
  [page 90] Check whether a term is a pid
- `is_port(Term) -> bool()`  
  [page 90] Check whether a term is a port
- `is_process_alive(Pid) -> bool()`  
  [page 90] Check whether a process is alive
- `is_record(Term, RecordTag) -> bool()`  
  [page 91] Check whether a term appears to be a record
- `is_record(Term, RecordTag, Size) -> bool()`: Check whether a term appears to be a record [page 91]
- `is_reference(Term) -> bool()`: Check whether a term is a reference [page 91]
- `is_tuple(Term) -> bool()`: Check whether a term is a tuple [page 91]
- `length(List) -> int()`: Length of a list [page 92]
- `link(Pid) -> true`: Create a link to another process (or port) [page 92]
- `list_to_atom(String) -> atom()`: Convert from text representation to an atom [page 92]
- `list_to_binary(IoList) -> binary()`: Convert a list to a binary [page 92]
- `list_to_existing_atom(String) -> atom()`: Convert from text representation to an atom [page 93]
- `list_to_float(String) -> float()`: Convert from text representation to a float [page 93]
- `list_to_integer(String) -> int()`: Convert from text representation to an integer [page 93]
- `erlang:list_to_integer(String, Base) -> int()`: Convert from text representation to an integer [page 93]
- `list_to_pid(String) -> pid()`: Convert from text representation to a pid [page 94]
- `list_to_tuple(List) -> tuple()`: Convert a list to a tuple [page 94]
- `load_module(Module, Binary) -> {module, Module} | {error, Reason}`: Load object code for a module [page 94]
- `erlang:loaded() -> [Module]`: List of all loaded modules [page 95]
- `erlang:localtime() -> {Date, Time}`: Current local date and time [page 95]
- `erlang:localtime_to_universaltime({Date1, Time1}) -> {Date2, Time2}`: Convert from local to Universal Time Coordinated (UTC) date and time [page 95]
- `erlang:localtime_to_universaltime({Date1, Time1}, IsDST) -> {Date2, Time2}`: Convert from local to Universal Time Coordinated (UTC) date and time [page 95]
- `make_ref() -> ref()`: Return an almost unique reference [page 96]
- `erlang:make_tuple(Arity, InitialValue) -> tuple()`: Create a new tuple of a given arity [page 96]
- `erlang:md5(Data) -> Digest`: Compute an MD5 message digest [page 96]
- `erlang:md5_final(Context) -> Digest`: Finish the update of an MD5 context and return the computed MD5 message digest [page 96]
- `erlang:md5_init()`: Create an MD5 context
  (Page 97)
- `erlang:md5_update(Context, Data)`: Update an MD5 context with data, and return a new context
  (Page 97)
- `erlang:memory()`: Information about dynamically allocated memory
  (Page 97)
- `erlang:memory(Type | [Type])`: Information about dynamically allocated memory
  (Page 98)
- `module_loaded(Module)`: Check if a module is loaded
  (Page 98)
- `erlang:monitor(Type, Item)`: Start monitoring
  (Page 99)
- `monitor_node(Node, Flag)`: Monitor the status of a node
  (Page 100)
- `node()`: Name of the local node
  (Page 100)
- `node(Arg)`: At which node is a pid, port or reference located
  (Page 101)
- `nodes()`: All visible nodes in the system
  (Page 101)
- `nodes(Arg | [Arg])`: All nodes of a certain type in the system
  (Page 101)
- `now()`: Elapsed time since 00:00 GMT
  (Page 101)
- `open_port(PortName, PortSettings)`: Open a port
  (Page 102)
- `erlang:phash(Term, Range)`: Portable hash function
  (Page 104)
- `erlang:phash2(Term [, Range])`: Portable hash function
  (Page 104)
- `pid_to_list(Pid)`: Text representation of a pid
  (Page 104)
- `port_close(Port)`: Close an open port
  (Page 105)
- `port_command(Port, Data)`: Send data to a port
  (Page 105)
- `port_connect(Port, Pid)`: Set the owner of a port
  (Page 105)
- `port_control(Port, Operation, Data)`: Perform a synchronous control operation on a port
  (Page 106)
- `erlang:port_call(Port, Operation, Data)`: Synchronous call to a port with term data
  (Page 106)
- `erlang:port_info(Port)`: Information about a port
  (Page 107)
- `erlang:port_info(Port, Item)`: Information about a port
  (Page 107)
Kernel Reference Manual

- `erlang:port_to_list(Port) -> string()` [page 107] Text representation of a port identifier
- `erlang:ports() -> [port()]` [page 108] All open ports
- `pre_loaded() -> [Module]` [page 108] List of all pre-loaded modules
- `erlang:process_display(Pid, Type) -> void()` [page 108] Write information about a local process on standard error
- `process_flag(Flag, Value) -> OldValue` [page 108] Set process flags for the calling process
- `process_flag(Pid, Flag, Value) -> OldValue` [page 109] Set process flags for a process
- `process_info(Pid) -> [{Item, Info}] | undefined` [page 109] Information about a process
- `process_info(Pid, Item) -> {Item, Info} | undefined | []` [page 110] Information about a process
- `processes() -> [pid()]` [page 111] All processes
- `purge_module(Module) -> void()` [page 111] Remove old code for a module
- `put(Key, Val) -> OldVal | undefined` [page 111] Add a new value to the process dictionary
- `erlang:raise(Class, Reason, Stacktrace)` [page 111] Stop execution with an exception of given class, reason and call stack
- `backtrace` [page 112] Stop execution with an exception of given class, reason and call stack
- `erlang:read_timer(TimerRef) -> int() | false` [page 112] Number of milliseconds remaining for a timer
- `erlang:ref_to_list(Ref) -> string()` [page 112] Text representation of a reference
- `register(RegName, Pid | Port) -> true` [page 113] Register a name for a pid (or port)
- `registered() -> [RegName]` [page 113] All registered names
- `erlang:resume_process(Pid) -> true` [page 113] Resume a suspended process
- `round(Number) -> int()` [page 113] Return an integer by rounding a number
- `self() -> pid()` [page 114] Pid of the calling process
- `erlang:send(Dest, Msg) -> Msg` [page 114] Send a message
- `erlang:send(Dest, Msg, [Option]) -> Res` [page 114] Send a message conditionally
- `erlang:send_after(Time, Dest, Msg) -> TimerRef` [page 115] Start a timer
- `erlang:send_nosuspend(Dest, Msg) -> bool()` [page 115] Try to send a message without ever blocking
- `erlang:send_nosuspend(Dest, Msg, Options) -> bool()`  
  [page 116] Try to send a message without ever blocking

- `erlang:set_cookie(Node, Cookie) -> true`  
  [page 116] Set the magic cookie of a node

- `setelement(Index, Tuple1, Value) -> Tuple2`  
  [page 117] Set Nth element of a tuple

- `size(Item) -> int()`  
  [page 117] Size of a tuple or binary

- `spawn(Fun) -> pid()`  
  [page 117] Create a new process with a fun as entry point

- `spawn(Node, Fun) -> pid()`  
  [page 117] Create a new process with a fun as entry point on a given node

- `spawn(Module, Function, Args) -> pid()`  
  [page 117] Create a new process with a function as entry point

- `spawn(Node, Module, Function, ArgumentList) -> pid()`  
  [page 118] Create a new process with a function as entry point on a given node

- `spawn_link(Fun) -> pid()`  
  [page 118] Create and link to a new process with a fun as entry point

- `spawn_link(Node, Fun) -> pid()`  
  [page 118] Create and link to a new process with a fun as entry point on a specified node

- `spawn_link(Module, Function, Args) -> pid()`  
  [page 118] Create and link to a new process with a function as entry point

- `spawn_link(Node, Module, Function, Args) -> pid()`  
  [page 119] Create and link to a new process with a function as entry point on a given node

- `spawn_opt(Fun, [Option]) -> pid()`  
  [page 119] Create a new process with a fun as entry point

- `spawn_opt(Node, Fun, [Option]) -> pid()`  
  [page 119] Create a new process with a fun as entry point on a given node

- `spawn_opt(Module, Function, Args, [Option]) -> pid()`  
  [page 119] Create a new process with a function as entry point

- `spawn_opt(Node, Module, Function, Args, [Option]) -> pid()`  
  [page 120] Create a new process with a function as entry point on a given node

- `split_binary(Bin, Pos) -> {Bin1, Bin2}`  
  [page 121] Split a binary into two

- `erlang:start_timer(Time, Dest, Msg) -> TimerRef`  
  [page 121] Start a timer

- `statistics(Type) -> Res`  
  [page 122] Information about the system

- `erlang:suspend_process(Pid) -> true`  
  [page 122] Suspend a process

- `erlang:system_flag(Flag, Value) -> OldValue`  
  [page 123] Set system flags

- `erlang:system_info(Type) -> Res`  
  [page 123] Information about the system
- `erlang:system_monitor()` -> `MonSettings`
  [page 126] Current system performance monitoring settings
- `erlang:system_monitor(undefined | {MonitorPid, Options})` -> `MonSettings`
  [page 127] Set or clear system performance monitoring options
- `erlang:system_monitor(MonitorPid, [Option])` -> `MonSettings`
  [page 127] Set system performance monitoring options
- `term_to_binary(Term)` -> `ext_binary()`
  [page 128] Encode a term to an Erlang external term format binary
- `term_to_binary(Term, [Option])` -> `ext_binary()`
  [page 128] Encode a term to an Erlang external term format binary
- `throw(Any)`
  [page 128] Throw an exception
- `time()` -> `{Hour, Minute, Second}`
  [page 128] Current time
- `tl(List1)` -> `List2`
  [page 129] Tail of a list
- `erlang:trace(PidSpec, How, FlagList)` -> `int()`
  [page 129] Set trace flags for a process or processes
- `erlang:trace_info(PidOrFunc, Item)` -> `Res`
  [page 132] Trace information about a process or function
- `erlang:trace_pattern(MFA, MatchSpec)` -> `int()`
  [page 133] Set trace patterns for global call tracing
- `erlang:trace_pattern(MFA, MatchSpec, FlagList)` -> `int()`
  [page 133] Set trace patterns for tracing of function calls
- `trunc(Number)` -> `int()`
  [page 135] Return an integer by truncating a number
- `tuple_to_list(Tuple)` -> `[term()]`
  [page 135] Convert a tuple to a list
- `erlang:universaltime()` -> `{Date, Time}`
  [page 135] Current date and time according to Universal Time Coordinated (UTC)
- `erlang:universaltime_to_localtime({Date1, Time1})` -> `{Date2, Time2}`
  [page 136] Convert from Universal Time Coordinated (UTC) to local date and time
- `unlink(Id)` -> `true`
  [page 136] Remove a link, if there is one, to another process or port
- `unregister(RegName)` -> `true`
  [page 137] Remove the registered name for a process (or port)
- `whereis(RegName)` -> `pid() | port() | undefined`
  [page 137] Get the pid (or port) with a given registered name
- `erlang:yield()` -> `true`
  [page 137] Let other processes get a chance to execute
error_handler

The following functions are exported:

- undefined_function(Module, Function, Args) -> term()
  [page 138] Called when an undefined function is encountered
- undefined_lambda(Module, Fun, Args) -> term()
  [page 138] Called when an undefined lambda (fun) is encountered

error_logger

The following functions are exported:

- error_msg(Format) -> ok
  [page 140] Send an standard error event to the error logger
- error_msg(Format, Data) -> ok
  [page 140] Send an standard error event to the error logger
- format(Format, Data) -> ok
  [page 140] Send an standard error event to the error logger
- error_report(Report) -> ok
  [page 141] Send a standard error report event to the error logger
- error_report(Type, Report) -> ok
  [page 141] Send a user defined error report event to the error logger
- warning_map() -> Tag
  [page 141] Return the current mapping for warning events
- warning_msg(Format) -> ok
  [page 142] Send a standard warning event to the error logger
- warning_msg(Format, Data) -> ok
  [page 142] Send a standard warning event to the error logger
- warning_report(Report) -> ok
  [page 142] Send a standard warning report event to the error logger
- warning_report(Type, Report) -> ok
  [page 142] Send a user defined warning report event to the error logger
- info_msg(Format) -> ok
  [page 143] Send a standard information event to the error logger
- info_msg(Format, Data) -> ok
  [page 143] Send a standard information event to the error logger
- info_report(Report) -> ok
  [page 143] Send a standard information report event to the error logger
- info_report(Type, Report) -> ok
  [page 143] Send a user defined information report event to the error logger
- add_report_handler(Handler) -> Result
  [page 144] Add an event handler to the error logger
- add_report_handler(Handler, Args) -> Result
  [page 144] Add an event handler to the error logger
- delete_report_handler(Handler) -> Result
  [page 144] Delete an event handler from the error logger
 tty(Flag) -> ok
 [page 144] Enable or disable printouts to the tty
 logfile(Request) -> ok | Filename | {error, What}
 [page 144] Enable or disable error printouts to a file

file

The following functions are exported:

• change_group(Filename, Gid) -> ok | {error, Reason}
 [page 146] Change group of a file
• change_owner(Filename, Uid) -> ok | {error, Reason}
 [page 146] Change owner of a file
• change_owner(Filename, Uid, Gid) -> ok | {error, Reason}
 [page 147] Change owner and group of a file
• change_time(Filename, Mtime) -> ok | {error, Reason}
 [page 147] Change the modification time of a file
• change_time(Filename, Mtime, Atime) -> ok | {error, Reason}
 [page 147] Change the modification and last access time of a file
• close(IoDevice) -> ok | {error, Reason}
 [page 147] Close a file
• consult(Filename) -> {ok, Terms} | {error, Reason}
 [page 147] Read Erlang terms from a file
• copy(Source, Destination) ->
 [page 148] Copy file contents
• copy(Source, Destination, ByteCount) -> {ok, BytesCopied} | {error, Reason}
 [page 148] Copy file contents
• del_dir(Dir) -> ok | {error, Reason}
 [page 148] Delete a directory
• delete(Filename) -> ok | {error, Reason}
 [page 149] Delete a file
• eval(Filename) -> ok | {error, Reason}
 [page 149] Evaluate Erlang expressions in a file
• eval(Filename, Bindings) -> ok | {error, Reason}
 [page 149] Evaluate Erlang expressions in a file
• file_info(Filename) -> {ok, FileInfo} | {error, Reason}
 [page 150] Get information about a file (deprecated)
• format_error(Reason) -> Chars
 [page 150] Return a descriptive string for an error reason
• get_cwd() -> {ok, Dir} | {error, Reason}
 [page 150] Get the current working directory
• get_cwd(Drive) -> {ok, Dir} | {error, Reason}
 [page 150] Get the current working directory for the drive specified
• list_dir(Dir) -> {ok, Filenames} | {error, Reason}
 [page 151] List files in a directory
- `make_dir(Dir) -> ok | {error, Reason}`
  [page 151] Make a directory

- `make_link(Existing, New) -> ok | {error, Reason}`
  [page 151] Make a hard link to a file

- `make_symlink(Name1, Name2) -> ok | {error, Reason}`
  [page 152] Make a symbolic link to a file or directory

- `open(Filename, Modes) -> {ok, IoDevice} | {error, Reason}`
  [page 152] Open a file

- `path_consult(Path, Filename) -> {ok, Terms, FullName} | {error, Reason}`
  [page 154] Read Erlang terms from a file

- `path_eval(Path, Filename) -> {ok, FullName} | {error, Reason}`
  [page 154] Evaluate Erlang expressions in a file

- `path_open(Path, Filename, Modes) -> {ok, IoDevice, FullName} | {error, Reason}`
  [page 155] Open a file

- `path_script(Path, Filename) -> {ok, Value, FullName} | {error, Reason}`
  [page 155] Evaluate and return the value of Erlang expressions in a file

- `path_script(Path, Filename, Bindings) -> {ok, Value, FullName} | {error, Reason}`
  [page 156] Evaluate and return the value of Erlang expressions in a file

- `pid2name(Pid) -> string() | undefined`
  [page 156] Return the name of the file handled by a pid

- `position(IoDevice, Location) -> {ok, NewPosition} | {error, Reason}`
  [page 156] Set position in a file

- `pread(IoDevice, LocNums) -> {ok, DataL} | {error, Reason}`
  [page 157] Read from a file at certain positions

- `pread(IoDevice, Location, Number) -> {ok, Data} | {error, Reason}`
  [page 157] Read from a file at a certain position

- `pwrite(IoDevice, LocBytes) -> ok | {error, {N, Reason}}`
  [page 157] Write to a file at certain positions

- `pwrite(IoDevice, Location, Bytes) -> ok | {error, Reason}`
  [page 158] Write to a file at a certain position

- `read(IoDevice, Number) -> {ok, Data} | eof | {error, Reason}`
  [page 158] Read from a file

- `read_file(Filename) -> {ok, Binary} | {error, Reason}`
  [page 158] Read a file

- `read_file_info(Filename) -> {ok, FileInfo} | {error, Reason}`
  [page 159] Get information about a file

- `read_link(Name) -> {ok, Filename} | {error, Reason}`
  [page 160] See what a link is pointing to

- `read_link_info(Name) -> {ok, FileInfo} | {error, Reason}`
  [page 160] Get information about a link or file

- `rename(Source, Destination) -> ok | {error, Reason}`
  [page 160] Rename a file
- **script**(Filename) -> {ok, Value} | {error, Reason}
  [page 161] Evaluate and return the value of Erlang expressions in a file
- **script**(Filename, Bindings) -> {ok, Value} | {error, Reason}
  [page 161] Evaluate and return the value of Erlang expressions in a file
- **set_cwd**(Dir) -> ok
  [page 162] Set the current working directory
- **sync**(IoDevice) -> ok | {error, Reason}
  [page 162] Synchronizes the in-memory state of a file with that on the physical medium
- **truncate**(IoDevice) -> ok | {error, Reason}
  [page 162] Truncate a file
- **write**(IoDevice, Bytes) -> ok | {error, Reason}
  [page 162] Write to a file
- **write_file**(Filename, Binary) -> ok | {error, Reason}
  [page 163] Write a file
- **write_file**(Filename, Binary, Modes) -> ok | {error, Reason}
  [page 163] Write a file
- **write_file_info**(Filename, FileInfo) -> ok | {error, Reason}
  [page 163] Change information about a file

**gen_tcp**

The following functions are exported:

- **connect**(Address, Port, Options) -> {ok, Socket} | {error, Reason}
  [page 169] Connect to a TCP port
- **connect**(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}
  [page 169] Connect to a TCP port
- **listen**(Port, Options) -> {ok, ListenSocket} | {error, Reason}
  [page 170] Set up a socket to listen on a port
- **accept**(ListenSocket) -> {ok, Socket} | {error, Reason}
  [page 170] Accept an incoming connection request on a listen socket
- **accept**(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}
  [page 170] Accept an incoming connection request on a listen socket
- **send**(Socket, Packet) -> ok | {error, Reason}
  [page 171] Send a packet
- **recv**(Socket, Length) -> {ok, Packet} | {error, Reason}
  [page 171] Receive a packet from a passive socket
- **recv**(Socket, Length, Timeout) -> {ok, Packet} | {error, Reason}
  [page 171] Receive a packet from a passive socket
- **controlling_process**(Socket, Pid) -> ok | {error, eperm}
  [page 171] Change controlling process of a socket
- **close**(Socket) -> ok | {error, Reason}
  [page 171] Close a TCP socket
- **shutdown**(Socket, How) -> ok | {error, Reason}
  [page 172] Immediately close a socket
gen_udp

The following functions are exported:

- `open(Port) -> {ok, Socket} | {error, Reason}`
  [page 173] Associate a UDP port number with the process calling it
- `open(Port, Options) -> {ok, Socket} | {error, Reason}`
  [page 173] Associate a UDP port number with the process calling it
- `send(Socket, Address, Port, Packet) -> ok | {error, Reason}`
  [page 174] Send a packet
- `recv(Socket, Length) -> {ok, {Address, Port, Packet}} | {error, Reason}`
  [page 174] Receive a packet from a passive socket
- `recv(Socket, Length, Timeout) -> {ok, {Address, Port, Packet}} | {error, Reason}`
  [page 174] Receive a packet from a passive socket
- `controlling_process(Socket, Pid) -> ok`  
  [page 174] Change controlling process of a socket
- `close(Socket) -> ok | {error, Reason}`  
  [page 175] Close a UDP socket

global

The following functions are exported:

- `del_lock(Id)`  
  [page 177] Delete a lock
- `del_lock(Id, Nodes) -> void()`  
  [page 177] Delete a lock
- `notify_all_name(Name, Pid1, Pid2) -> none`  
  [page 177] Name resolving function that notifies both pids
- `random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`  
  [page 177] Name resolving function that kills one pid
- `random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`  
  [page 177] Name resolving function that notifies one pid
- `register_name(Name, Pid)`  
  [page 178] Globally register a name for a pid
- `register_name(Name, Pid, Resolve) -> yes | no`  
  [page 178] Globally register a name for a pid
- `registered_names() -> [Name]`  
  [page 178] All globally registered names
- `re_register_name(Name, Pid)`  
  [page 178] Atomically re-register a name
- `re_register_name(Name, Pid, Resolve) -> void()`  
  [page 178] Atomically re-register a name
- `send(Name, Msg) -> Pid`  
  [page 179] Send a message to a globally registered pid
- set_lock(Id)
  [page 179] Set a lock on the specified nodes
- set_lock(Id, Nodes)
  [page 179] Set a lock on the specified nodes
- set_lock(Id, Nodes, Retries) -> boolean()
  [page 179] Set a lock on the specified nodes
- sync() -> void()
  [page 180] Synchronize the global name server
- trans(Id, Fun)
  [page 180] Micro transaction facility
- trans(Id, Fun, Nodes)
  [page 180] Micro transaction facility
- trans(Id, Fun, Nodes, Retries) -> Res | aborted
  [page 180] Micro transaction facility
- unregister_name(Name) -> void()
  [page 180] Remove a globally registered name for a pid
- whereis_name(Name) -> pid() | undefined
  [page 180] Get the pid with a given globally registered name

global_group
The following functions are exported:

- global_groups() -> {GroupName, GroupNames} | undefined
  [page 182] Return the global group names
- info() -> [[Item, Info]]
  [page 182] Information about global groups
- monitor_nodes(Flag) -> ok
  [page 182] Subscribe to node status changes
- own_nodes() -> Nodes
  [page 182] Return the group nodes
- registered_names(Where) -> Names
  [page 183] Return globally registered names
- send(Name, Msg) -> pid() | {badarg, {Name, Msg}}
  [page 183] Send a message to a globally registered pid
- send(Where, Name, Msg) -> pid() | {badarg, {Name, Msg}}
  [page 183] Send a message to a globally registered pid
- sync() -> ok
  [page 183] Synchronize the group nodes
- whereis_name(Name) -> pid() | undefined
  [page 183] Get the pid with a given globally registered name
- whereis_name(Where, Name) -> pid() | undefined
  [page 183] Get the pid with a given globally registered name
heart

The following functions are exported:

- `set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}`
  [page 186] Set a temporary reboot command
- `clear_cmd() -> ok`
  [page 186] Clear the temporary boot command
- `get_cmd() -> {ok, Cmd}`
  [page 186] Get the temporary reboot command

inet

The following functions are exported:

- `close(Socket) -> ok`
  [page 188] Close a socket of any type
- `get_rc() -> [{Par, Val}]`
  [page 188] Return a list of IP configuration parameters
- `format_error(Posix) -> string()`
  [page 189] Return a descriptive string for an error reason
- `getaddr(Host, Family) -> {ok, Address} | {error, posix()}`
  [page 189] Return the IP-address for a host
- `getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}`
  [page 189] Return the IP-addresses for a host
- `gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}`
  [page 189] Return a hostent record for the host with the given address
- `gethostbyname(Name) -> {ok, Hostent} | {error, posix()}`
  [page 189] Return a hostent record for the host with the given name
- `gethostbyname(Name, Family) -> {ok, Hostent} | {error, posix()}`
  [page 189] Return a hostent record for the host with the given name
- `gethostname() -> {ok, Hostname} | {error, posix()}`
  [page 189] Return the local hostname
- `getopts(Socket, Options) -> OptionValues`
  [page 190] Get one or more options for a socket
- `peername(Socket) -> {ok, {Address, Port}} | {error, posix()}`
  [page 190] Return the address and port for the other end of a connection
- `port(Socket) -> {ok, Port}`
  [page 190] Return the local port number for a socket
- `sockname(Socket) -> {ok, {Address, Port}} | {error, posix()}`
  [page 190] Return the local address and port number for a socket
- `setopts(Socket, Options) -> ok | {error, posix()}`
  [page 190] Set one or more options for a socket
init

The following functions are exported:

- `{boot(BootArgs) -> void()}
  [page 196] Start the Erlang runtime system
- `{get_args() -> [Arg]}
  [page 196] Get all non-flag command line arguments
- `{get_argument(Flag) -> {ok, Arg} | error}
  [page 196] Get the values associated with a command line user flag
- `{get_arguments() -> Flags}
  [page 197] Get all command line user flags
- `{get_plain_arguments() -> [Arg]}
  [page 197] Get all non-flag command line arguments
- `{get_status() -> {InternalStatus, ProvidedStatus}}
  [page 197] Get system status information
- `{reboot() -> void()}
  [page 198] Take down an Erlang node smoothly
- `{restart() -> void()}
  [page 198] Restart the running Erlang node
- `{script_id() -> Id}
  [page 198] Get the identity of the used boot script
- `{stop() -> void()}
  [page 198] Take down an Erlang node smoothly

net_adm

The following functions are exported:

- `{dns_hostname(Host) -> {ok, Name} | {error, Host}}
  [page 201] Official name of a host
- `{host_file() -> Hosts | {error, Reason}}
  [page 201] Read the .hosts.erlang file
- `{localhost() -> Name}
  [page 201] Name of the local host
- `{names() -> {ok, [[Name, Port]]} | {error, Reason}}
  [page 201] Names of Erlang nodes at a host
- `{names(Host) -> {ok, [[Name, Port]]} | {error, Reason}}
  [page 201] Names of Erlang nodes at a host
- `{ping(Node) -> pong | pang}
  [page 202] Set up a connection to a node
- `{world() -> [node()]}
  [page 202] Lookup and connect to all nodes at all hosts in .hosts.erlang
- `{world(Arg) -> [node()]}
  [page 202] Lookup and connect to all nodes at all hosts in .hosts.erlang
- `{world_list(Hosts) -> [node()]}
  [page 202] Lookup and connect to all nodes at specified hosts
- `{world_list(Hosts, Arg) -> [node()]}
  [page 202] Lookup and connect to all nodes at specified hosts
net_kernel

The following functions are exported:

- allow(Nodes) -> ok | error
  [page 204] Limit access to a specified set of nodes
- connect_node(Node) -> true | false | ignored
  [page 204] Establish a connection to a node
- monitor_nodes(Flag) -> ok | ignored | Error
  [page 205] Subscribe to node status change messages
- monitor_nodes(Flag, Options) -> ok | ignored | Error
  [page 205] Subscribe to node status change messages
- get_net_ticktime() -> Res
  [page 206] Get net_ticktime
- set_net_ticktime(NetTicktime) -> Res
  [page 206] Set net_ticktime
- set_net_ticktime(NetTicktime, TransitionPeriod) -> Res
  [page 206] Set net_ticktime
- start([Name]) -> {ok, pid()} | {error, Reason}
  [page 207] Turn an Erlang runtime system into a distributed node
- start([Name, NameType]) -> {ok, pid()} | {error, Reason}
  [page 207] Turn an Erlang runtime system into a distributed node
- start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}
  [page 207] Turn an Erlang runtime system into a distributed node
- stop() -> ok | {error, not_allowed | not_found}
  [page 207] Turn a node into a non-distributed Erlang runtime system

OS

The following functions are exported:

- cmd(Command) -> string()
  [page 208] Execute a command in a shell of the target OS
- find_executable(Name) -> Filename | false
  [page 208] Absolute filename of a program
- find_executable(Name, Path) -> Filename | false
  [page 208] Absolute filename of a program
- getenv() -> [string()]
  [page 208] List all environment variables
- getenv(VarName) -> Value | false
  [page 209] Get the value of an environment variable
- getpid() -> Value
  [page 209] Return the process identifier of the emulator process
- putenv(VarName, Value) -> true
  [page 209] Set a new value for an environment variable
- type() -> {Osfamily, Osname} | Osfamily
  [page 209] Return the OS family and, in some cases, OS name of the current operating system
- version() -> {Major, Minor, Release} | VersionString
  [page 209] Return the Operating System version
packages

The following functions are exported:

- no functions exported
  [page 213] x

pg2

The following functions are exported:

- create(Name) -> void()
  [page 214] Create a new, empty process group
- delete(Name) -> void()
  [page 214] Delete a process group
- get_closest_pid(Name) -> Pid | {error, Reason}
  [page 214] Common dispatch function
- get_members(Name) -> [Pid] | {error, Reason}
  [page 215] Return all processes in a group
- get_local_members(Name) -> [Pid] | {error, Reason}
  [page 215] Return all local processes in a group
- join(Name, Pid) -> ok | {error, Reason}
  [page 215] Join a process to a group
- leave(Name, Pid) -> ok | {error, Reason}
  [page 215] Make a process leave a group
- which_groups() -> [Name]
  [page 215] Return a list of all known groups
- start()
  [page 215] Start the pg2 server
- start_link() -> {ok, Pid} | {error, Reason}
  [page 215] Start the pg2 server

rpc

The following functions are exported:

- call(Node, Module, Function, Args) -> Res | {badrpc, Reason}
  [page 216] Evaluate a function call on a node
- call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}
  [page 216] Evaluate a function call on a node
- block_call(Node, Module, Function, Args) -> Res | {badrpc, Reason}
  [page 216] Evaluate a function call on a node in the RPC server's context
- block_call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}
  [page 217] Evaluate a function call on a node in the RPC server's context
- async_call(Node, Module, Function, Args) -> Key
  [page 217] Evaluate a function call on a node, asynchronous version
- `yield(Key) -> Res | {badrpc, Reason}`
  [page 217] Deliver the result of evaluating a function call on a node (blocking)
- `nb_yield(Key) -> {value, Val} | timeout`
  [page 217] Deliver the result of evaluating a function call on a node (non-blocking)
- `nb_yield(Key, Timeout) -> {value, Val} | timeout`
  [page 218] Deliver the result of evaluating a function call on a node (non-blocking)
- `multicall(Module, Function, Args) -> {ResL, BadNodes}`
  [page 218] Evaluate a function call on a number of nodes
- `multicall(Nodes, Module, Function, Args) -> {ResL, BadNodes}`
  [page 218] Evaluate a function call on a number of nodes
- `multicall(Module, Function, Args, Timeout) -> {ResL, BadNodes}`
  [page 218] Evaluate a function call on a number of nodes
- `multicall(Nodes, Module, Function, Args, Timeout) -> {ResL, BadNodes}`
  [page 218] Evaluate a function call on a number of nodes
- `cast(Node, Module, Function, Args) -> void()`
  [page 219] Run a function on a node ignoring the result
- `eval_everywhere(Module, Function, Args) -> void()`
  [page 219] Run a function on all nodes, ignoring the result
- `eval_everywhere(Nodes, Module, Function, Args) -> void()`
  [page 219] Run a function on specific nodes, ignoring the result
- `abcast(Name, Msg) -> void()`
  [page 220] Broadcast a message asynchronously to a registered process on all nodes
- `abcast(Nodes, Name, Msg) -> void()`
  [page 220] Broadcast a message asynchronously to a registered process on specific nodes
- `sbcast(Name, Msg) -> {GoodNodes, BadNodes}`
  [page 220] Broadcast a message synchronously to a registered process on all nodes
- `sbcast(Nodes, Name, Msg) -> {GoodNodes, BadNodes}`
  [page 220] Broadcast a message synchronously to a registered process on specific nodes
- `server_call(Node, Name, ReplyWrapper, Msg) -> Reply | {error, Reason}`
  [page 220] Interact with a server on a node
- `multi_server_call(Name, Msg) -> {Replies, BadNodes}`
  [page 221] Interact with the servers on a number of nodes
- `multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`
  [page 221] Interact with the servers on a number of nodes
- `safe_multi_server_call(Name, Msg) -> {Replies, BadNodes}`
  [page 221] Interact with the servers on a number of nodes (deprecated)
- `safe_multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`
  [page 221] Interact with the servers on a number of nodes (deprecated)
- `parallel_eval(FuncCalls) -> ResL`
  [page 222] Evaluate several function calls on all nodes in parallel
- `pmap({Module, Function}, ExtraArgs, List2) -> List1`
  [page 222] Parallel evaluation of mapping a function over a list
Kernel Reference Manual

- pinfo(Pid) \rightarrow [[\text{Item}, \text{Info}]] | \text{undefined}
  [page 222] Information about a process
- pinfo(Pid, Item) \rightarrow \{\text{Item}, \text{Info}\} | \text{undefined} | []
  [page 222] Information about a process

seq_trace

The following functions are exported:

- set_token(Token) \rightarrow \text{PreviousToken}
  [page 223] Set the trace token
- set_token(Component, Val) \rightarrow \{Component, OldVal\}
  [page 223] Set a component of the trace token
- get_token() \rightarrow \text{TraceToken}
  [page 224] Return the value of the trace token
- get_token(Component) \rightarrow \{Component, Val\}
  [page 224] Return the value of a trace token component
- print(TraceInfo) \rightarrow \text{void()}
  [page 224] Put the Erlang term TraceInfo into the sequential trace output
- print(Label, TraceInfo) \rightarrow \text{void()}
  [page 224] Put the Erlang term TraceInfo into the sequential trace output
- reset_trace() \rightarrow \text{void()}
  [page 225] Stop all sequential tracing on the local node
- set_system_tracer(Tracer) \rightarrow \text{OldTracer}
  [page 225] Set the system tracer
- get_system_tracer() \rightarrow \text{Tracer}
  [page 225] Return the pid() or port() of the current system tracer.

user

No functions are exported.

wrap_log_reader

The following functions are exported:

- chunk(Continuation)
  [page 232] Read a chunk of objects written to a wrap log.
- chunk(Continuation, N) \rightarrow \{Continuation2, Terms\} | \{Continuation2, Terms, Badbytes\} | \{Continuation2, eof\} | \{error, Reason\}
  [page 232] Read a chunk of objects written to a wrap log.
- close(Continuation) \rightarrow \text{ok}
  [page 233] Close a log
- open(Filename) \rightarrow \text{OpenRet}
  [page 233] Open a log file
- open(Filename, N) \rightarrow \text{OpenRet}
  [page 233] Open a log file
zlib

The following functions are exported:

- **open()** -> Z  
  [page 235] Open a stream and return a stream reference
- **close(Z)** -> ok  
  [page 235] Close a stream
- **deflateInit(Z)** -> ok  
  [page 235] Initialize a session for compression
- **deflateInit(Z, Level)** -> ok  
  [page 235] Initialize a session for compression
- **deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy)** -> ok  
  [page 235] Initialize a session for compression
- **deflate(Z, Data)** -> Compressed  
  [page 236] Compress data
- **deflate(Z, Data, Flush)** ->  
  [page 236] Compress data
- **deflateSetDictionary(Z, Dictionary)** -> Adler32  
  [page 237] Initialize the compression dictionary
- **deflateReset(Z)** -> ok  
  [page 237] Reset the deflate session
- **deflateParams(Z, Level, Strategy)** -> ok  
  [page 237] Dynamically update deflate parameters
- **deflateEnd(Z)** -> ok  
  [page 238] End deflate session
- **inflateInit(Z)** -> ok  
  [page 238] Initialize a session for decompression
- **inflateInit(Z, WindowBits)** -> ok  
  [page 238] Initialize a session for decompression
- **inflate(Z, Data)** -> DeCompressed  
  [page 238] Decompress data
- **inflateSetDictionary(Z, Dictionary)** -> ok  
  [page 238] Initialize the decompression dictionary
- **inflateReset(Z)** -> ok  
  [page 239] Reset the inflate session
- **inflateEnd(Z)** -> ok  
  [page 239] End inflate session
- **setBufSize(Z, Size)** -> ok  
  [page 239] Set buffer size
- **getBufSize(Z)** -> Size  
  [page 239] Get buffer size
- **crc32(Z)** -> CRC  
  [page 240] Get current CRC
- **crc32(Z, Binary)** -> CRC  
  [page 240] Calculate CRC
- `crc32(Z, PrevCRC, Binary) -> CRC`
  [page 240] Calculate CRC

- `adler32(Z, Binary) -> Checksum`
  [page 240] Calculate the adler checksum

- `adler32(Z, PrevAdler, Binary) -> Checksum`
  [page 240] Calculate the adler checksum

- `compress(Binary) -> Compressed`
  [page 241] Compress a binary with standard zlib functionality

- `uncompress(Binary) -> Decompressed`
  [page 241] Uncompress a binary with standard zlib functionality

- `zip(Binary) -> Compressed`
  [page 241] Compress a binary without the zlib headers

- `unzip(Binary) -> Decompressed`
  [page 241] Uncompress a binary without the zlib headers

- `gzip(Data) -> Compressed`
  [page 241] Compress a binary with gz header

- `gunzip(Bin) -> Decompressed`
  [page 241] Uncompress a binary with gz header

**app**

No functions are exported.

**config**

No functions are exported.
The Kernel application is the first application started. It is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STD LIB. The Kernel application contains the following services:

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

Error Logger Event Handlers

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

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

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

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

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

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

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

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

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

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

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

`error_logger` Specifies when error reports are written to a file. 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` Defines global groups, see global_group(3).

- `GroupTuple = {GroupName, [Node]} | {GroupName, PublishType, [Node]}
- GroupName = atom()
- PublishType = normal | hidden
- Node = node()`
inet.default.connect.options = [[Opt, Val]] Specifies default options for connect sockets, see inet(3).

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

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

{inet.dist.listen_min, First} See below.

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

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

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

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

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

The time $T$, in which a node that is not responding is detected, is calculated as:

$$\text{MinT} < T < \text{MaxT}$$

where:

$$\text{MinT} = \text{TickTime} - \text{TickTime} / 4$$
$$\text{MaxT} = \text{TickTime} + \text{TickTime} / 4$$

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

Note: All communicating nodes should have the same TickTime value specified. Note: Normally, a terminating node is detected immediately.

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

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

sync.nodes.timeout = integer() | infinity Specifies the amount of time (in milliseconds) this node will wait for the mandatory and optional nodes to start. If this parameter is undefined, no node synchronization is performed. This option also makes sure that global is synchronized.
start_ddll = true | false 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 Starts the dist_ac server if the parameter is true.
   This parameter should be set to true for systems that use distributed applications.
   The default value is false. If this parameter is undefined, the server is started if the parameter distributed is set.

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

boot_server_slaves = [SlaveIP] If the start_boot_server configuration parameter is true, this parameter can be used to initialize boot_server with a list of slave IP addresses. SlaveIP = string() | atom |
   {integer(),integer(),integer(),integer()} where 0 < integer() <=255.
   Examples of SlaveIP in atom, string and tuple form are:
   '150.236.16.70', "150,236,16,70", {150,236,16,70}.
   The default value is [].

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

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

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

shutdown_func = {Mod, Func} Where:
   - Mod = atom()
   - Func = atom()

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

See Also

app(4) [page 242], application(3) [page 34], code(3) [page 45], disk_log(3) [page 53],
erl_boot_server(3) [page 67], erl_ddll(3) [page 69], error_logger(3) [page 140], file(3) [page 146], global(3) [page 176], global_group(3) [page 181], heart(3) [page 185],
inet(3) [page 187], net_kernel(3) [page 204], os(3) [page 208], pg2(3) [page 214],
rpc(3) [page 216], seq_trace(3) [page 223], user(3) [page 231]
application

Erlang Module

In OTP, application denotes a component implementing some specific functionality, that can be started and stopped as a unit, and which can be re-used in other systems as well. This module interfaces the application controller, a process started at every Erlang runtime system, and contains functions for controlling applications (for example starting and stopping applications), and functions to access information about applications (for example configuration parameters).

A n application is defined by an application specification. The specification is normally located in an application resource file called Application.app, where Application is the name of the application. Refer to app(4) [page 242] for more information about the application specification.

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

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

Exports

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

Types:
• Application = atom()
• Env = [{Par,Val}]
• Par = atom()
• Val = term()

Returns the configuration parameters and their values for Application. If the argument is omitted, it defaults to the application of the calling process.

If the specified application is not loaded, or if the process executing the call does not belong to any application, the function returns [].

get_all_key() -> {ok, Keys} | []
get_all_key(Application) -> {ok, Keys} | undefined

Types:
• Application = atom()
• Keys = [{Key,Val}]
Returns the application specification keys and their values for Application. If the argument is omitted, it defaults to the application of the calling process. If the specified application is not loaded, the function returns undefined. If the process executing the call does not belong to any application, the function returns [].

get_application() -> {ok, Application} | undefined
get_application(Pid | Module) -> {ok, Application} | undefined

Types:
- Pid = pid()
- Module = atom()
- Application = atom()

Returns the name of the application to which the process Pid or the module Module belongs. Providing no argument is the same as calling get_application(self()). If the specified process does not belong to any application, or if the specified process or module does not exist, the function returns undefined.

get_env(Par) -> {ok, Val} | undefined
get_env(Application, Par) -> {ok, Val} | undefined

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

Returns the value of the configuration parameter Par for Application. If the application argument is omitted, it defaults to the application of the calling process. If the specified application is not loaded, or the configuration parameter does not exist, or if the process executing the call does not belong to any application, the function returns undefined.

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

Types:
- Application = atom()
- Key = atom()
- Val = term()

Returns the value of the application specification key Key for Application. If the application argument is omitted, it defaults to the application of the calling process. If the specified application is not loaded, or the specification key does not exist, or if the process executing the call does not belong to any application, the function returns undefined.

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

Types:
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:
 starts the permission for Application to run at the current node. The application must have been loaded using load/1,2 for the function to have effect.
If the permission of a loaded, but not started, application is set to false, start will return ok but the application will not be started until the permission is set to true.
If the permission of a running application is set to false, the application will be stopped. If the permission later is set to true, it will be restarted.
If the application is distributed, setting the permission to false means that the application will be started at, or moved to, another node according to how its distribution is configured (see load/2 above).
The function does not return until the application is started, stopped or successfully moved to another node. However, in some cases where permission is set to true the function may return ok even though the application itself has not started. This is true when an application cannot start because it has dependencies to other applications which have not yet been started. When they have been started, Application will be started as well.
By default, all applications are loaded with permission true on all nodes. The permission is configurable by using the Kernel configuration parameter permissions.

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

Types:
- Application = atom()
- Par = atom()
- Val = term()
- Timeout = int() | infinity
Sets the value of the configuration parameter Par for Application.
set_env/3 uses the standard gen_server timeout value (5000 ms). A Timeout argument can be provided if another timeout value is useful, for example, in situations where the application controller is heavily loaded.

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

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

Types:
- Application = atom()
- Type = permanent | transient | temporary
Reason = term()

Starts Application. If it is not loaded, the application controller will first load it using load/1. It will make sure any included applications are loaded, but will not start them. That is assumed to be taken care of in the code for Application.

The application controller checks the value of the application specification key applications, to ensure that all applications that should be started before this application are running. If not, {error,{not_started,App}} is returned, where App is the name of the missing application.

The application controller then creates an application master for the application. The application master is the group leader of all the processes in the application. The application master starts the application by calling the application callback function Module:start/2 as defined by the application specification key mod.

The Type argument specifies the type of the application. If omitted, it defaults to temporary.

- If a permanent application terminates, all other applications and the entire Erlang node are also terminated.
- If a transient application terminates with Reason == normal, this is reported but no other applications are terminated. If a transient application terminates abnormally, all other applications and the entire Erlang node are also terminated.
- If a temporary application terminates, this is reported but no other applications are terminated.

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

Note also that the transient type is of little practical use, since when a supervision tree terminates, the reason is set to shutdown, not normal.

```erlang
start_type() -> StartType | local | undefined
Types:
- StartType = normal | {takeover,Node} | {failover,Node}
- Node = node()
```

This function is intended to be called by a process belonging to an application, when the application is being started, to determine the start type which is either StartType or local.

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

local is returned if only parts of the application is being restarted (by a supervisor), or if the function is called outside a startup.

If the process executing the call does not belong to any application, the function returns undefined.

```erlang
stop(Application) -> ok | {error, Reason}
Types:
- Application = atom()
- Reason = term()
```
Stops Application. The application master calls `Module:prep_stop/1`, if such a function is defined, and then tells the top supervisor of the application to shutdown (see `supervisor(3)`). This means that the entire supervision tree, including included applications, is terminated in reversed start order. After the shutdown, the application master calls `Module:stop/1`. `Module` is the callback module as defined by the application specification key `mod`.

Last, the application master itself terminates. Note that all processes with the application master as group leader, i.e. processes spawned from a process belonging to the application, thus are terminated as well.

When stopped, the application is still loaded.

In order to stop a distributed application, `stop/1` has to be called on all nodes where it can execute (that is, on all nodes where it has been started). The call to `stop/1` on the node where the application currently executes will stop its execution. The application will not be moved between nodes due to `stop/1` being called on the node where the application currently executes before `stop/1` is called on the other nodes.

```erlang
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`.
```

```erlang
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.
```

```erlang
unset_env(Application, Par) -> ok
unset_env(Application, Par, Timeout) -> ok
Types:
  • Application = atom()
  • Par = atom()
```

Kernel Reference Manual application

Kernel Application (KERNEL) 39
which_applications() -> [{Application, Description, Vsn}]
which_applications(Timeout) -> [{Application, Description, Vsn}]

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

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

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

CALLBACK MODULE

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

Exports

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

Types:
- StartType = normal | {takeover, Node} | {failover, Node}
- Node = node()
- StartArgs = term()
- Pid = pid()
- State = term()
This function is called whenever an application is started using application:start/1,2, and should start the processes of the application. If the application is structured according to the OTP design principles as a supervision tree, this means starting the top supervisor of the tree.

StartType defines the type of start:

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

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

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

Types:
- Phase = atom()
- StartType = normal | {takeover, Node} | {failover, Node}
- Node = node()
- PhaseArgs = term()
- Pid = pid()
- State = state()

This function is used to start an application with included applications, when there is a need for synchronization between processes in the different applications during startup. The start phases is defined by the application specification key start_phases == [{Phase, PhaseArgs}]. For included applications, the set of phases must be a subset of the set of phases defined for the including application.

The function is called for each start phase (as defined for the primary application) for the primary application and all included applications, for which the start phase is defined. See Module:start/2 for a description of StartType.

Module:prep_stop(State) -> NewState

Types:
- State = NewState = term()
This function is called when an application is about to be stopped, before shutting down the processes of the application.

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

The function is optional. If it is not defined, the processes will be terminated and then Module:stop(State) is called.

Module:stop(State)

Types:
- State = term()

This function is called whenever an application has stopped. It is intended to be the opposite of Module:start/2 and should do any necessary cleaning up. The return value is ignored.

State is the return value of Module:prep_stop/1, if such a function exists. Otherwise State is taken from the return value of Module:start/2.

Module:config_change(Changed, New, Removed) -> ok

Types:
- Changed = [{Par,Val}]
- New = [{Par,Val}]
- Removed = [Par]
- Par = atom()
- Val = term()

This function is called by an application after a code replacement, if there are any changes to the configuration parameters.

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

SEE ALSO

[OTP Design Principles], kernel(6) [page 30], app(4) [page 242]
auth

Erlang Module

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

Exports

**is_auth(Node)** -> yes | no

Types:
- Node = node()

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

Use net_adm:ping(Node) [page 202] instead.

**cookie()** -> Cookie

Types:
- Cookie = atom()

Use erlang:get_cookie() [page 85] instead.

**cookie(TheCookie)** -> true

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

Use erlang:set_cookie(node(), Cookie) [page 116] instead.

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

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

Equivalent to node_cookie(Node, Cookie) [page 43].

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

Types:
- Node = node()
- `Cookie = atom()`

  Sets the magic cookie of `Node` to `Cookie`, and verifies the status of the authorization. Equivalent to calling `erlang:set_cookie(Node, Cookie)` [page 116], followed by `auth:is_auth(Node)` [page 43].
This module contains the interface to the Erlang code server, which deals with the loading of compiled code into a running Erlang runtime system.

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

```
erl -mode interactive
```

Default mode is interactive.

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

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

**Code Path**

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

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

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

Application resource files (`.app` files) are also stored in the code path cache. This feature is used by the application controller (see application(3) [page 34]) to load applications efficiently in large systems.

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

Current and Old Code

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

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

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

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

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

Exports

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

Types:
- `Path = [Dir]`
- `Dir = string()`
- `What = bad, directory | bad, path`
Sets the code path to the list of directories Path.
Returns true if successful, or {error, bad_directory} if any Dir is not the name of a directory, or {error, bad_path} if the argument is invalid.

get_path() -> Path
Types:
- Path = [Dir]
- Dir = string()
Returns the code path

add_path(Dir) -> true | {error, What}
add_pathz(Dir) -> true | {error, What}
Types:
- Dir = string()
- What = bad_directory
Adds Dir to the code path. The directory is added as the last directory in the new path. If Dir already exists in the path, it is not added.
Returns true if successful, or {error, bad_directory} if Dir is not the name of a directory.

add_patha(Dir) -> true | {error, What}
Types:
- Dir = string()
- What = bad_directory
Adds Dir to the beginning of the code path. If Dir already exists, it is removed from the old position in the code path.
Returns true if successful, or {error, bad_directory} if Dir is not the name of a directory.

add_paths(Dirs) -> ok
add_pathsz(Dirs) -> ok
Types:
- Dirs = [Dir]
- Dir = string()
Adds the directories in Dirs to the end of the code path. If a Dir already exists, it is not added. This function always returns ok, regardless of the validity of each individual Dir.

add_pathsa(Dirs) -> ok
Types:
- Dirs = [Dir]
- Dir = string()
Adds the directories in Dirs to the beginning of the code path. If a Dir already exists, it is removed from the old position in the code path. This function always returns ok, regardless of the validity of each individual Dir.

\[\text{addpath}(\text{Name} | \text{Dir}) \rightarrow \text{true} | \text{false} | \{\text{error}, \text{What}\}\]

Types:
- \(\text{Name} = \text{atom}()\)
- \(\text{Dir} = \text{string}()\)
- \(\text{What} = \text{bad_name}\)

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

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

\[\text{replacepath}(\text{Name}, \text{Dir}) \rightarrow \text{true} | \{\text{error}, \text{What}\}\]

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

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

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

\[\text{loadfile}(\text{Module}) \rightarrow \{\text{module, Module}\} | \{\text{error, What}\}\]

Types:
- \(\text{Module} = \text{atom}()\)
- \(\text{What} = \text{nofile} | \text{sticky_directory} | \text{badarg} | \text{term}()\)

Tries to load the Erlang module \(\text{Module}\), using the code path. It looks for the object code file with an extension that corresponds to the Erlang machine used, for example \(\text{Module.beam}\). The loading fails if the module name found in the object code differs from the name Module. load_binary/3 [page 49] must be used to load object code with a module name that is different from the file name.

Returns \{module, Module\} if successful, or \{error, nofile\} if no object code is found, or \{error, sticky_directory\} if the object code resides in a sticky directory, or \{error, badarg\} if the argument is invalid. Also if the loading fails, an error tuple is returned. See erlang:load_module/2 [page 94] for possible values of what.

\[\text{loadabs}(\text{Filename}) \rightarrow \{\text{module, Module}\} | \{\text{error, What}\}\]

Types:
- \(\text{Filename} = \text{string}()\)
ensure_loaded(Module) -> {module, Module} | {error, What}

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

Tries to to load a module in the same way as load_file/1 [page 48]. In embedded mode, however, it does not load a module which is not already loaded, but returns {error, embedded} instead.

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

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

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

Returns {module, Module} if successful, or {error, sticky_directory} if the object code resides in a sticky directory, or {error, badarg} if any argument is invalid. Also if the loading fails, an error tuple is returned. See erlang:load_module/2 [page 94] for possible values of What.

delete(Module) -> true | false

Types:
- Module = atom()

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

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

purge(Module) -> true | false

Types:
- Module = atom()
Purges the code for Module, that is, removes code marked as old. If some processes still linger in the old code, these processes are killed before the code is removed.

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

**soft_purge(Module)** -> true | false

Types:
- Module = atom()

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

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

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

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

Checks if Module is loaded. If it is, {file, Loaded} is returned, otherwise false. Normally, Loaded is the absolute file name Absname from which the code was obtained. If the module is preloaded (see [script(4)]), Loaded==preloaded. If the module is Cover compiled (see [cover(3)]), Loaded==cover_compiled.

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

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

Returns a list of tuples {Module, Loaded} for all loaded modules. Loaded is normally the absolute file name, as described for is_loaded/1 [page 50].

**which(Module)** -> Which

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

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

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

Types:
- Module = atom()
- Binary = binary()
- Filename = string()

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

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

**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, $ROOT/lib, where $ROOT is the root directory of Erlang/OTP.

**lib_dir(Name)** → Dir | {error, What}

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

Searches the code path for a directory named .../Name[-Vsn]/ebin and returns the directory .../Name[-Vsn]. Mainly intended for finding the library directory of the Erlang/OTP application called Name.

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

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

**compiler_dir()** → Dir

Types:
- Dir = string()

Searches the code path for the compiler library directory. Equivalent of calling code:lib_dir(compiler).

**priv_dir(Name)** → PrivDir | {error, What}

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

Searches the code path for a directory named .../Name[-Vsn]/ebin and returns the directory .../Name[-Vsn]/priv. It is not checked if this directory really exists. Mainly intended for finding the priv directory of the Erlang/OTP application called Name.

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

`objfile_extension()` -> Ext

Types:
- Ext = string()

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

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

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

This function marks `Dir` as sticky.

Returns `ok` if successful, and an error tuple otherwise.

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

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

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

Returns `ok` if successful, and an error tuple otherwise.

`rehash()` -> ok

This function creates or rehashes the code path cache.

`where_is_file(Filename)` -> Absname | non_existing

Types:
- Filename = Absname = string()

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

`clash()` -> ok

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

Erlang Module

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

Two formats of the log files are supported, the internal format and the external format. The internal format supports automatic repair of log files that have not been properly closed, and makes it possible to efficiently read logged items in chunks using a set of functions defined in this module. In fact, this is the only way to read internally formatted logs. The external format leaves it up to the user to read the logged deep byte lists. The disk log module cannot repair externally formatted logs. An item logged to an internally formatted log must not occupy more than 4 GB of disk space (the size must fit in 4 bytes).

For each open disk log there is one process that handles requests made to the disk log; the disk log process is created when open/1 is called, provided there exists no process handling the disk log. A process that opens a disk log can either be an owner or an anonymous user of the disk log. Each owner is linked to the disk log process, and the disk log is closed by the owner should the owner terminate. Owners can subscribe to notifications, messages of the form [disk_log, Node, Log, Info] that are sent from the disk log process when certain events occur, see the commands below and in particular the open/1 option notify [page 63]. 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. A synchronous functions do not wait for the disk log process to actually write the items to the file, but return the control to the caller more or less immediately.

When using the internal format for logs, the functions log/2, log_terms/2, alog/2, and alog_terms/2 should be used. These functions log one or more Erlang terms. By prefixing each of the functions with a b (for “binary”) we get the corresponding blog functions for the external format. These functions log one or more deep lists of bytes or, alternatively, binaries of deep lists of bytes. For example, to log the string "hello" in ASCII format, we can use disk_log:blog(Log, "hello"), or disk_log:blog(Log, list_to_binary("hello")). The two alternatives are equally efficient. The blog
functions can be used for internally formatted logs as well, but in this case they must be called with binaries constructed with calls to term_to_binary/1. There is no check to ensure this; it is entirely the responsibility of the caller. If these functions are called with binaries that do not correspond to Erlang terms, the chunk/2, 3 and automatic repair functions will fail. The corresponding terms (not the binaries) will be returned when chunk/2, 3 is called.

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

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

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

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

The disk log module itself does not report errors to the error_logger module; it is up to the caller to decide whether the error logger should be employed or not. The function format_error/1 can be used to produce readable messages from error replies.

Information events are however sent to the error logger in two situations, namely when a log is repaired, or when a file is missing while reading chunks.

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

accessible_logs() -> [{LocalLog], [DistributedLog]}
  Types:
  • LocalLog = DistributedLog = term()
  The accessible_logs/0 function returns the names of the disk logs accessible on the current node. The first list contains local disk logs, and the second list contains distributed disk logs.

alog(Log, Term) balog(Log, Bytes) -> ok | {error, Reason}
  Types:
  • Log = term()
  • Term = term()
  • Bytes = binary() | [Byte]
  • Byte = [Byte] | 0 <= integer() <= 255
  • Reason = no_such_log
  The alog/2 and balog/2 functions asynchronously append an item to a disk log. The function alog/2 is used for internally formatted logs, and the function balog/2 for externally formatted logs. balog/2 can be used for internally formatted logs as well provided the binary was constructed with a call to term_to_binary/1.

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

alog_terms(Log, TermList) balog_terms(Log, BytesList) -> ok | {error, Reason}
  Types:
  • Log = term()
  • TermList = [term()]
  • BytesList = [Bytes]
  • Bytes = binary() | [Byte]
  • Byte = [Byte] | 0 <= integer() <= 255
  • Reason = no_such_log
  The alog_terms/2 and balog_terms/2 functions asynchronously append a list of items to a disk log. The function alog_terms/2 is used for internally formatted logs, and the function balog_terms/2 for externally formatted logs. balog_terms/2 can be used for internally formatted logs as well provided the binaries were constructed with calls to term_to_binary/1.

  The owners that subscribe to notifications will receive the message read_only, blocked_log or format_external in case the items cannot be written on the log, and possibly one or more of the messages wrap, full and error_status if items were written on the log. The message error_status is sent if there is something wrong with the header function or a file error occurred.
block(Log)
block(Log, QueueLogRecords) -> ok | {error, Reason}

Types:
- Log = term()
- QueueLogRecords = bool()
- Reason = no_such_log | nonode | {blocked_log, Log}

With a call to block/1,2 a process can block a log. If the blocking process is not an owner of the log, a temporary link is created between the disk log process and the blocking process. The link is used to ensure that the disk log is unblocked should the blocking process terminate without first closing or unblocking the log.

Any process can probe a blocked log with info/1 or close it with close/1. The blocking process can also use the functions chunk/2,3, bchunk/2,3, chunk_step/3, and unblock/1 without being affected by the block. Any other attempt than those hitherto mentioned to update or read a blocked log suspends the calling process until the log is unblocked or returns an error message {blocked_log, Log}, depending on whether the value of QueueLogRecords is true or false. The default value of QueueLogRecords is true, which is used by block/1.

change_header(Log, Header) -> ok | {error, Reason}

Types:
- Log = term()
- Header = {head, Head} | {head_func, {M,F,A}}
- Head = none | term() | binary() | [Byte]
- Byte = [Byte] | 0 <= integer() <= 255
- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {badarg, head}

The change_header/2 function changes the value of the head or head_func option of a disk log.

change_notify(Log, Owner, Notify) -> ok | {error, Reason}

Types:
- Log = term()
- Owner = pid()
- Notify = bool()
- Reason = no_such_log | nonode | {blocked_log, Log} | {badarg, notify} | {not_owner, Owner}

The change_notify/3 function changes the value of the notify option for an owner of a disk log.

change_size(Log, Size) -> ok | {error, Reason}

Types:
- Log = term()
- Size = integer() > 0 | infinity | {MaxNoBytes, MaxNoFiles}
- MaxNoBytes = integer() > 0
- MaxNoFiles = integer() > 0
The `change_size/2` function changes the size of an open log. For a halt log it is always possible to increase the size, but it is not possible to decrease the size to something less than the current size of the file.

For a wrap log it is always possible to increase both the size and number of files, as long as the number of files does not exceed 65000. If the maximum number of files is decreased, the change will not be valid until the current file is full and the log wraps to the next file. The redundant files will be removed next time the log wraps around, i.e. starts to log to file number 1.

As an example, assume that the old maximum number of files is 10 and that the new maximum number of files is 6. If the current file number is not greater than the new maximum number of files, the files 7 to 10 will be removed when file number 6 is full and the log starts to write to file number 1 again. Otherwise the files greater than the current file will be removed when the current file is full (e.g. if the current file is 8, the files 9 and 10); the files between new maximum number of files and the current file (i.e. files 7 and 8) will be removed next time file number 6 is full.

If the size of the files is decreased the change will immediately affect the current log. It will not of course change the size of log files already full until next time they are used. If the log size is decreased for instance to save space, the function `inc_wrap_file/1` can be used to force the log to wrap.

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

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

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

The first time `chunk` (or `bchunk`) is called, an initial continuation, the atom `start`, must be provided. If there is a disk log process running on the current node, terms are read from that log, otherwise an individual distributed log on some other node is chosen, if such a log exists.
When \texttt{chunk/3} is called, \( N \) controls the maximum number of terms that are read from the log in each chunk. Default is \texttt{infinity}, which means that all the terms contained in the 64 kilobyte chunk are read. If less than \( N \) terms are returned, this does not necessarily mean that the end of the file has been reached.

The \texttt{chunk} function returns a tuple \{\texttt{Continuation2, Terms}\}, where \texttt{Terms} is a list of terms found in the log. \texttt{Continuation2} is yet another continuation which must be passed on to any subsequent calls to \texttt{chunk}. With a series of calls to \texttt{chunk} it is possible to extract all terms from a log.

The \texttt{chunk} function returns a tuple \{\texttt{Continuation2, Terms, Badbytes}\} if the log is opened in read-only mode and the read chunk is corrupt. \texttt{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 \{\texttt{corrupt_log_file, FileName}\} is returned if the read chunk is corrupt.

\texttt{chunk} returns \texttt{eof} when the end of the log is reached, or \{\texttt{error, Reason}\} if an error occurs. Should a wrap log file be missing, a message is output on the error log.

When \texttt{chunk/2,3} is used with wrap logs, the returned continuation may or may not be valid in the next call to \texttt{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.

\begin{verbatim}
chunk_info(Continuation) -> InfoList | {error, Reason}
Types:
  • Continuation = cont()
  • Reason = \{no_continuation, Continuation\}

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

  • \{node, Node\}. Terms are read from the disk log running on \texttt{Node}.
\end{verbatim}

\begin{verbatim}
chunk_step(Log, Continuation, Step) -> ok, Continuation2 | {error, Reason}
Types:
  • Log = term()
  • Continuation = start \| cont()
  • Step = integer()
  • Continuation2 = cont()
  • Reason = no_such_log \| end_of_log \| \{format_external, Log\} \| \{blocked_log, Log\} \| \{badarg, continuation\} \| \{file_error, FileName, FileError\}

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

If the atom \texttt{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.

\(\text{close(Log)} \rightarrow \text{ok | \{error, Reason\}}\)

Types:
- \(\text{Reason} = \text{no_such_log | nonode | \{file_error, FileName, FileError\}}\)

The function \text{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.

\(\text{format_error(Error)} \rightarrow \text{Chars}\)

Types:
- \(\text{Chars} = [\text{char()} | \text{Chars}]\)

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

\(\text{inc_wrap_file(Log)} \rightarrow \text{ok | \{error, Reason\}}\)

Types:
- \(\text{Reason} = \text{no_such_log | nonode | \{read_only_mode, Log\} | \{blocked_log, Log\} | \{halt_log, Log\} | \{invalid_header, InvalidHeader\} | \{file_error, FileName, FileError\}}\)

The \text{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 \text{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 \text{wrap} message, but in case of an error with a reason tag of \text{invalid_header} or \text{file_error} an \text{error_status} message will be sent.

\(\text{info(Log)} \rightarrow \text{InfoList | \{error, no_such_log\}}\)

The \text{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 \text{open/1} option name.
- \{file, File\}. For halt logs File is the filename, and for wrap logs File is the base name.
diskLog

- \{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 63].
- \{status, Status\}, where Status is ok or \{blocked, QueueLogRecords\} as set by the functions block/1,2 and unblock/1.
- \{node, Node\}. The information returned by the current invocation of the info/1 function has been gathered from the disk log process running on Node.
- \{distributed, Dist\}. If the log is local on the current node, then Dist has the value local, otherwise all nodes where the log is distributed are returned as a list.

The following pairs are returned for all logs opened in read_write mode:

- \{head, Head\}. Depending of the value of the open/1 options head and head_func or set by the function change_header/2, the value of Head is none (default), \{head, H\} (head option) or \{M,F,A\} (head_func option).
- \{no_written_items, NoWrittenItems\}, where NoWrittenItems is the number of items written to the log since the disk log process was created.

The following pair is returned for halt logs opened in read_write mode:

- \{full, Full\}, where Full is true or false depending on whether the halt log is full or not.

The following pairs are returned for wrap logs opened in read_write mode:

- \{no_current_bytes, integer() \geq 0\} is the number of bytes written to the current wrap log file.
- \{no_current_items, integer() \geq 0\} is the number of items written to the current wrap log file, header inclusive.
- \{no_items, integer() \geq 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 \texttt{chunk/2,3, bchunk/2,3, and chunk\_step/3} functions do not affect any value returned by \texttt{info/1}.

\texttt{lclose(Log)}
\texttt{lclose(Log, Node) \rightarrow ok | \{error, Reason\}}

Types:
- \texttt{Node = node()}
- \texttt{Reason = no\_such\_log | \{file\_error, FileName, FileError\}}

The function \texttt{lclose/1} closes a local log or an individual distributed log on the current node. The function \texttt{lclose/2} closes an individual distributed log on the specified node if the node is not the current one. \texttt{lclose(Log)} is equivalent to \texttt{lclose(Log, node())}. See also \texttt{close/1} [page 59].

If there is no log with the given name on the specified node, \texttt{no\_such\_log} is returned.

\texttt{log(Log, Term)}
\texttt{blog(Log, Bytes) \rightarrow ok | \{error, Reason\}}

Types:
- \texttt{Log = term()}
- \texttt{Term = term()}
- \texttt{Bytes = binary() | [Byte]}
- \texttt{Byte = [Byte] | 0 <= integer() <= 255}
- \texttt{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 \texttt{log/2} and \texttt{blog/2} functions synchronously append a term to a disk log. They return \texttt{ok} or \texttt{\{error, Reason\}} when the term has been written to disk. If the log is distributed, \texttt{ok} is always returned, unless all nodes are down. Terms are written by means of the ordinary \texttt{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 \texttt{sync/1} function must be called.

The \texttt{log/2} function is used for internally formatted logs, and \texttt{blog/2} for externally formatted logs. \texttt{blog/2} can be used for internally formatted logs as well provided the binary was constructed with a call to \texttt{term\_to\_binary/1}.

The owners that subscribe to notifications will be notified of an error with an \texttt{error\_status} message if the error reason tag is \texttt{invalid\_header} or \texttt{file\_error}.

\texttt{log\_terms(Log, TermList)}
\texttt{blog\_terms(Log, BytesList) \rightarrow ok | \{error, Reason\}}

Types:
- \texttt{Log = term()}
- \texttt{TermList = [term()]}
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.

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 \texttt{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 \texttt{<base_name>.N}, where \texttt{N} is an integer. Each wrap log will also have two files called \texttt{<base_name>.idx} and \texttt{<base_name>.siz}.

- \{linkto, LinkTo\}. If \texttt{LinkTo} is a pid, that pid becomes an owner of the log. If \texttt{LinkTo} is none the log records that it is used anonymously by some process by incrementing the \texttt{users} counter. By default, the process which calls \texttt{open/1} owns the log.

- \{repair, Repair\}. If \texttt{Repair} is true, the current log file will be repaired, if needed. If 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 \texttt{truncate} is given, the log file will be truncated, creating an empty log. \texttt{Repair} is true, which has no effect on logs opened in read-only mode.

- \{type, Type\} is the type of the log. Default is \texttt{halt}.

- \{format, Format\} specifies the format of the disk log. Default is \texttt{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 \texttt{infinity}, which for halt implies that there is no maximum size. For wrap logs, the \texttt{Size} parameter may be either a pair \{MaxNoBytes, MaxNoFiles\} or \texttt{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 \texttt{MaxNoBytes} bytes on each file and use \texttt{MaxNoFiles} files before starting all over with the first wrap log file. Regardless of \texttt{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 \texttt{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. \texttt{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 \texttt{NoLostItems} is the number of items written on the log since the disk log process was created. For wrap logs \texttt{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. \texttt{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. \texttt{Items} is the items from the log attempt.
- `{disk_log, Node, Log, {format, external, Items}}` is sent when `alog/2` or `alog_terms/2` is used for internally formatted logs. Items is the items from the log attempt.
- `{disk_log, Node, Log, full}` is sent when an attempt to log items to a wrap log would write more bytes than the limit set by the `size` option.
- `{disk_log, Node, Log, {error_status, Status}}` is sent when the error status changes. The error status is defined by the outcome of the last attempt to log items to a the log or to truncate the log or the last use of `sync/1`, `inc_wrap_file/1` or `change_size/2`. Status is one of `ok` and `{error, Error}`, the former being the initial value.

- `{head, Head}` specifies a header to be written first on the log file. If the log is a wrap log, the item `Head` is written first in each new file. `Head` should be a term if the format is `internal`, and a deep list of bytes (or a binary) otherwise. Default is `none`, which means that no header is written first on the file.

- `{head_func, {M, F, A}}` specifies a function to be called each time a new log file is opened. The call `M:F(A)` is assumed to return `{ok, Head}`. The item `Head` is written first in each file. `Head` should be a term if the format is `internal`, and a deep list of bytes (or a binary) otherwise.

- `{mode, Mode}` specifies if the log is to be opened in read-only or read-write mode. It defaults to `read_write`.

The `open/1` function returns `{ok, Log}` if the log file was successfully opened. If the file was successfully repaired, the tuple `{repaired, Log, {recovered, Rec}, {badbytes, Bad}}` is returned, where `Rec` is the number of whole Erlang terms found in the file and `Bad` is the number of bytes in the file which were non-Erlang terms. If the distributed parameter was given, `open/1` returns a list of successful replies and a list of erroneous replies. Each reply is tagged with the node name. When a disk log is opened in read-write mode, any existing log file is checked for. If there is none a new empty log is created, otherwise the existing file is opened at the position after the last logged item, and the logging of items will commence from there. If the format is `internal` and the existing file is not recognized as an internally formatted log, a tuple `{error, {not_a_log_file, FileName}}` is returned.

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

If a log with a given name is local on some node, and one tries to open the log distributed on the same node, then the tuple `{error, {node_already_open, Name}}` is returned. The same tuple is returned if the log is distributed on some node, and one tries to open the log locally on the same node. Opening individual distributed disk logs for the first time adds those logs to a (possibly empty) distributed disk log. The option values supplied are used on all nodes mentioned by the distributed option. Individual distributed logs know nothing about each other's option values, so each node can be given unique option values by creating a distributed log with several calls to `open/1`.

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

If an attempt to open a log file for the first time fails, the disk log process terminates with the EXIT message {{failed,Reason},[disk_log,open,1]}. The function returns [error, Reason] for all other errors.

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

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

The pid2name/1 function returns the name of the log given the pid of a disk log process on the current node, or undefined if the given pid is not a disk log process.

This function is meant to be used for debugging only.

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

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

The reopen functions first rename the log file to File and then re-create a new log file.
In case of a wrap log, File is used as the base name of the renamed files. By default the header given to open/1 is written first in the newly opened log file, but if the Head or the BHead argument is given, this item is used instead. The header argument is used only once; 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.
The `truncate/1,2` 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}}}`.

The `unblock/1` function unblocks a log. A log can only be unblocked by the blocking process.

**See Also**

`file(3)` [page 146], `pg2(3)` [page 214], `wrap_log_reader(3)` [page 232]
**erl_boot_server**

Erlang Module

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

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

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

**Exports**

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

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

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

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

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

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

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

Types:
- `Slave = Host`
- `Host = atom()`
- `What = term()`

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

```
delete_slave(Slave) -> ok | {error, What}
```
Types:
- Slave = Host
- Host = atom()
- What = void()

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

\[ \text{which_slaves()} \rightarrow \text{Slaves} \]

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

Returns the current list of allowed slave hosts.

**SEE ALSO**

init(3) [page 196], erl_prj_loader(3) [page 71]
erl_ddll

Erlang Module

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

Exports

\[\text{load\_driver(Path, Name)} \rightarrow \text{ok} | \text{error, ErrorDesc}\]

Types:
- \(\text{Path} = \text{Name} = \text{string()} \lor \text{atom()}\)
- \(\text{ErrorDesc} = \text{term()}\)

Loads and links the dynamic driver \(\text{Name}\). \(\text{Path}\) is a file path to the directory containing the driver. \(\text{Name}\) must be a sharable object/dynamic library. Two drivers with different \(\text{Paths}\) cannot be loaded under the same name. The number of dynamically loadable drivers is limited by the size of \text{driver\_tab} in \text{driver\_tab.c}.

\[\text{unload\_driver(Name)} \rightarrow \text{ok} | \text{error, ErrorDesc}\]

Types:
- \(\text{Name} = \text{string()} \lor \text{atom()}\)
- \(\text{Reason} = \text{term()}\)

Unloads the dynamic driver \(\text{Name}\). This will fail if any port programs are running the code that is being unloaded. Linked-in drivers cannot be unloaded. The process must previously have called \text{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.

\[\text{loaded\_drivers()} \rightarrow \{\text{ok, Drivers}\}\]

Types:
- \(\text{Drivers} = \{\text{Driver()}\}\)
- \(\text{Driver} = \text{string()}\)

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

\[\text{format\_error(ErrorDesc)} \rightarrow \text{string()}\]

Types:
- \(\text{ErrorDesc} - \text{see below}\)
Takes an ErrorDesc returned by load_driver/2 or unload_driver/1 and returns a string which describes the error or warning.

SEE ALSO

erl_driver(4), driver_entry(4)
erl_prim_loader

Erlang Module

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

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

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

Exports

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

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

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

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

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

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

get_file(File) -> {ok, Bin, FullName} | error
Types:
- File = string()
- Bin = binary()
- FullName = string()

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

get_path() -> {ok, Path}

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

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

set_path(Path) -> ok

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

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

Protocol

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_file</td>
<td>[102</td>
<td>FileName]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[122]</td>
</tr>
<tr>
<td>stop</td>
<td>eof</td>
<td>terminate</td>
</tr>
</tbody>
</table>
Command Line Flags

The `erl_prim_loader` module interprets the following command line flags:

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

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

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

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

SEE ALSO

`init(3)` [page 196], `erl_boot_server(3)` [page 67]
By convention, most built-in functions (BIFs) are seen as being in the module `erlang`. A number of the BIFs are viewed more or less as part of the Erlang programming language and are auto-imported. Thus, it is not necessary to specify the module name and both the calls `atom_to_list(Erlang)` and `erlang:atom_to_list(Erlang)` are identical.

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

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

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

### DATA TYPES

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

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

### Exports

- **abs(Number) -> int() | float()**
  - Types:
    - `Number = number()`
  - Returns an integer or float which is the arithmetical absolute value of `Number`.
    - `> abs(-3.33).
      3.33000`
    - `> abs(-3).
      3`

  - Allowed in guard tests.

- **erlang:append_element(Tuple1, Term) -> Tuple2**
Types:
- Tuple1 = Tuple2 = tuple()
- Term = term()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Failure: `{error, handler:undefined, function/3}` is called if the applied function is not exported. The error handler can be redefined (see `process_flag`[2] [page 108]). If the error handler is undefined, or if the user has redefined the default `error_handler` so the replacement module is undefined, an error with the reason `undef` is generated.
atom_to_list(Atom) \rightarrow \text{string()}

Types:
- Atom = \text{atom()}

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

```erlang
goto(latex{atom_to_list('Erlang'). "Erlang"})
goto(basic)
```

binary_to_list(Binary) \rightarrow \text{[char()]}  

Types:
- Binary = \text{binary()}

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

```erlang
goto(basic)
```

binary_to_list(Binary, Start, Stop) \rightarrow \text{[char()]}  

Types:
- Binary = \text{binary()}
- Start = Stop = 1..size(Binary)

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

```erlang
goto(basic)
```

binary_to_term(Binary) \rightarrow \text{term()}

Types:
- Binary = \text{ext_binary()}

Returns an Erlang term which is the result of decoding the binary object Binary, which must be encoded according to the Erlang external term format. See also \text{term_to_binary/1} [page 128].

```erlang
goto(basic)
```

erlang:bump_reductions(Reductions) \rightarrow \text{void()}

Types:
- Reductions = \text{int()}

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

**Warning:**

This BIF might be removed in a future version of the Beam machine without prior warning. It is unlikely to be implemented in other Erlang implementations.

```erlang
goto(basic)
```

erlang:cancel_timer(TimerRef) \rightarrow \text{Time | false}

Types:
Kernel Reference Manual

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

Cancels a timer, where TimerRef was returned by either erlang:send_after/3 [page 115] or erlang:start_timer/3 [page 121]. If the timer is there to be removed, the function returns the time in milliseconds left until the timer would have expired, otherwise false (which means that TimerRef was never a timer, that it has already been cancelled, or that it has already delivered its message).

See also erlang:send_after/3 [page 115], erlang:start_timer/3 [page 121], and erlang:read_timer/1 [page 112].

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

cancel_timer(TimerRef)

check_process_code(Pid, Module) -> bool()

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

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

> check_process_code(Pid, lists).
false

See also code(3) [page 45].

concat_binary(ListOfBinaries)

Do not use; use list_to_binary/1 [page 92] instead.

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

Types:
- Year = Month = Day = int()

Returns the current date as {Year, Month, Day}. The time zone and daylight saving time correction depend on the underlying OS.

> date()
{1995, 2, 19}

delete_module(Module) -> true | undefined

Types:
- Module = atom()
Makes the current code for Module become old code, and deletes all references for this module from the export table. Returns undefined if the module does not exist, otherwise true.

**Warning:**
This BIF is intended for the code server (see code(3) [page 45]) and should not be used elsewhere.

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

erlang:demonitor(MonitorRef) -> true

**Types:**
- MonitorRef = ref()

If MonitorRef is a reference which the calling process obtained by calling erlang:monitor/2 [page 99], this monitoring is turned off. If the monitoring is already turned off, nothing happens.

Once erlang:demonitor(MonitorRef) has returned it is guaranteed that no `{'DOWN', MonitorRef, _, _, _}` message due to the monitor will be placed in the callers message queue in the future. A `{'DOWN', MonitorRef, _, _, _}` message might have been placed in the callers message queue prior to the call, though. Therefore, in most cases, it is advisable to remove such a 'DOWN' message from the message queue after monitoring has been stopped. erlang:demonitor(MonitorRef, [flush]) [page 78] can be used instead of erlang:demonitor(MonitorRef) if this cleanup is wanted.

**Note:**
Prior to OTP release R11B (erts version 5.5) erlang:demonitor/1 behaved completely asynchronously, i.e., the monitor was active until the “demonitor signal” reached the monitored entity. This had one undesirable effect, though. You could never know when you were guaranteed not to receive a DOWN message due to the monitor.

Current behavior can be viewed as two combined operations: asynchronously send a “demonitor signal” to the monitored entity and ignore any future results of the monitor.

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

erlang:demonitor(MonitorRef, OptionList) -> true

**Types:**
- MonitorRef = ref()
- OptionList = [Option]
- Option = flush
erlang:demonitor(MonitorRef, []) is equivalent to erlang:demonitor(MonitorRef) [page 78].

Currently the following Options are valid:

flush Remove (one) `{DOWN', MonitorRef, _, _, _}` message, if there is one, from
the callers message queue after monitoring has been stopped.
Calling erlang:demonitor(MonitorRef, [flush]) is equivalent to:

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

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

Failure: badarg if OptionList is not a list, or if Option is not a valid option, or the same
failure as for erlang:demonitor/1 [page 78]

disconnect_node(Node) -> bool() | ignored
Types:
  • Node = atom()
Forces the disconnection of a node. This will appear to the node Node as if the local
node has crashed. This BIF is mainly used in the Erlang network authentication
protocols. Returns true if disconnection succeeds, otherwise false. If the local node is
not alive, the function returns ignored.

erlang:display(Term) -> true
Types:
  • Term = term()
Prints a text representation of Term on the standard output.

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

element(N, Tuple) -> term()
Types:
  • N = 1..size(Tuple)
  • Tuple = tuple()
Returns the Nth element (numbering from 1) of Tuple.
element(2, \{a, b, c\}).

Allowed in guard tests.

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

Types:
- Key = Val = term()

Returns the process dictionary and deletes it.

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

erase(Key) -> Val | undefined

Types:
- Key = Val = term()

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

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

erlang:erl()

Types:
- Reason = term()

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

> catch erlang:erl()
    \{'EXIT',\{foobar,\{eri_eval,do_apply,5\},
        \{eri_eval,expr,5\},
        \{shell,exprs,6\},
        \{shell,eval_loop,3\}\}\}

erlang:erl(Reason, Args)

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

**exit(Reason)**

Types:
- \texttt{Reason = term()}

Stops the execution of the calling process with the exit reason \texttt{Reason}, where \texttt{Reason} is any term. Since evaluating this function causes the process to terminate, it has no return value.

\begin{verbatim}
> exit(foobar).
** exited: foobar **
> catch exit(foobar).
{"EXIT", foobar}
\end{verbatim}

**exit(Pid, Reason) \rightarrow true**

Types:
- \texttt{Pid = pid()}
- \texttt{Reason = term()}

Sends an exit signal with exit reason \texttt{Reason} to the process \texttt{Pid}.

The following behavior apply if \texttt{Reason} is any term except \texttt{normal} or \texttt{kill}:
- If \texttt{Pid} is not trapping exits, \texttt{Pid} itself will exit with exit reason \texttt{Reason}. If \texttt{Pid} is trapping exits, the exit signal is transformed into a message \{'EXIT', \texttt{From}, \texttt{Reason}\} and delivered to the message queue of \texttt{Pid}. \texttt{From} is the pid of the process which sent the exit signal. See also process\_flag/2 [page 108].
- If \texttt{Reason} is the atom \texttt{normal}, \texttt{Pid} will not exit. If it is trapping exits, the exit signal is transformed into a message \{'EXIT', \texttt{From}, \texttt{normal}\} and delivered to its message queue.
- If \texttt{Reason} is the atom \texttt{kill}, that is if exit(Pid, \texttt{kill}) is called, an untrappable exit signal is sent to \texttt{Pid} which will unconditionally exit with exit reason \texttt{killed}.

**erlang:fault(Reason)**

Types:
- \texttt{Reason = term()}

Stops the execution of the calling process with the reason \texttt{Reason}. This is an old equivalent to erlang\_error(Reason) [page 80].

**erlang:fault(Reason, Args)**

Types:
- \texttt{Reason = term()}
- \texttt{Args = [term()]}
Stops the execution of the calling process with the reason \texttt{Reason}. This is an old equivalent to \texttt{erlang:error(Reason, Args)} [page 80].

\texttt{float(Number) -> float()}

Types:
\begin{itemize}
\item \texttt{Number = number()}
\end{itemize}

Returns a float by converting \texttt{Number} to a float.

\begin{verbatim}
> float(55).
55.0000
\end{verbatim}

Allowed in guard tests.

\textbf{Note:}

Note that if used on the top-level in a guard, it will test whether the argument is a floating point number; for clarity, use \texttt{is_float/1} [page 89] instead.

When \texttt{float/1} is used in an expression in a guard, such as \texttt{float(A) == 4.0}, it converts a number as described above.

\texttt{float_to_list(Float) -> string()}

Types:
\begin{itemize}
\item \texttt{Float = float()}
\end{itemize}

Returns a string which corresponds to the text representation of \texttt{Float}.

\begin{verbatim}
> float_to_list(7.0).
"7.00000000000000000000e+00"
\end{verbatim}

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

Types:
\begin{itemize}
\item \texttt{Fun = fun()}
\item \texttt{Item, Info – see below}
\end{itemize}

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

\textbf{Warning:}

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

There are two types of funs with slightly different semantics:
A fun created by fun M:F/A is called an external fun. Calling it will always call the function F with arity A in the latest code for module M. Note that module M does not even need to be loaded when the fun fun M:F/A is created.

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

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

- **{type, Type}** Type is either local or external.
- **{module, Module}** Module (an atom) is the module name.
  - If Fun is a local fun, Module is the module in which the fun is defined.
  - If Fun is an external fun, Module is the module that the fun refers to.
- **{name, Name}** Name (an atom) is a function name.
  - If Fun is a local fun, Name is the name of the local function that implements the fun. (This name was generated by the compiler, and is generally only of informational use. As it is a local function, it is not possible to call it directly.) If no code is currently loaded for the fun, [] will be returned instead of an atom.
  - If Fun is an external fun, Name is the name of the exported function that the fun refers to.
- **{arity, Arity}** Arity is the number of arguments that the fun should be called with.
- **{env, Env}** Env (a list) is the environment or free variables for the fun. (For external funs, the returned list is always empty.)

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

- **{pid, Pid}** Pid is the pid of the process that originally created the fun.
- **{index, Index}** Index (an integer) is an index into the module's fun table.
- **{new_index, Index}** Index (an integer) is an index into the module's fun table.
- **{new_uniq, Uniq}** Uniq (a binary) is a unique value for this fun.
- **{uniq, Uniq}** Uniq (an integer) is a unique value for this fun.

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

Types:

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

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

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

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

See erlang:fun_info/1 [page 82].

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

Types:

- Fun = fun()
Returns a string which corresponds to the text representation of Fun.

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

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

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

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

This function is retained mainly for backwards compatibility.

garbage_collect() -> true

Forces an immediate garbage collection of the currently executing process. The function should not be used, unless it has been noticed – or there are good reasons to suspect – that the spontaneous garbage collection will occur too late or not at all. Improper use may seriously degrade system performance.

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

garbage_collect(Pid) -> bool()

Types:
- Pid = pid()

Works like erlang:garbage_collect() but on any process. The same caveats apply.

Returns false if Pid refers to a dead process; true otherwise.

get() -> [{Key, Val}]

Types:
- Key = Val = term()

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

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

[{{key1,merry},{key2,lambs},{key3,{are,playing}}]

get(Key) -> Val | undefined

Types:
- Key = Val = term()

Returns the value Val associated with Key in the process dictionary, or undefined if Key does not exist.
erlang:get_cookie() -> Cookie | nocookie

Types:
- Cookie = atom()

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

get_keys(Val) -> [Key]

Types:
- Val = Key = term()

Returns a list of keys which are associated with the value Val in the process dictionary.

erlang:get_stacktrace() -> [{Module, Function, Arity | Args}]

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

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

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

The stacktrace is the same data as the catch operator returns, for example:

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

See also erlang:error/1 [page 80] and erlang:error/2 [page 80].

group_leader() -> GroupLeader

Types:
- GroupLeader = pid()
Returns the pid of the group leader for the process which evaluates the function.

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

group_leader(GroupLeader, Pid) -> true
Types:
- GroupLeader = Pid = pid()
Sets the group leader of Pid to GroupLeader. Typically, this is used when a processes started from a certain shell should have another group leader than init.
See also group_leader/0 [page 85].

halt()
Halt the Erlang runtime system and indicates normal exit to the calling environment. Has no return value.

> halt(),
os_prompt%

halt(Status)
Types:
- Status = int()>=0 | string()
Status must be a non-negative integer, or a string. Halts the Erlang runtime system. Has no return value. If Status is an integer, it is returned as an exit status of Erlang to the calling environment. If Status is a string, produces an Erlang crash dump with String as slogan, and then exits with a non-zero status code.

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

erlang:hash(Term, Range) -> Hash
Returns a hash value for Term within the range 1..Range. The allowed range is 1..2^27-1.

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

hd(List) -> term()
Types:
- List = [term()]
  Returns the head of List, that is, the first element.

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

Allowed in guard tests.
Failure: badarg if List is the empty list [].

erlang:hibernate(Module, Function, Args)

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

Puts the calling process into a wait state where its memory allocation has been reduced as much as possible, which is useful if the process does not expect to receive any messages in the near future.

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

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

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

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

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

erlang:info(Type) -> Res

This BIF is now equivalent to erlang:system_info/1 [page 123].

integer_to_list(Integer) -> string()

Types:
- Integer = int()

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

> integer_to_list(77).
"77"
erlang:integer_to_list(Integer, Base) -> string()

Types:
• Integer = int()
• Base = 2..36

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

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

iolist_to_binary(IoListOrBinary) -> binary()

Types:
• IoListOrBinary = iolist() | binary()

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

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

iolist_size(Item) -> int()

Types:
• Item = iolist() | binary()

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

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

is_alive() -> bool()

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

is_atom(Term) -> bool()

Types:
• Term = term()

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

Allowed in guard tests.

is_binary(Term) -> bool()

Types:
- Term = term()
  Returns true if Term is a binary; otherwise returns false.
  Allowed in guard tests.

\textbf{is\_boolean(Term) \to bool()}

\textbf{Types:}
- Term = term()
  Returns true if Term is either the atom \texttt{true} or the atom \texttt{false} (i.e. a boolean); otherwise returns false.
  Allowed in guard tests.

\textbf{erlang:is\_builtin(Module, Function, Arity) \to bool()}

\textbf{Types:}
- Module = Function = atom()
- Arity = int()
  Returns true if Module:Function/Arity is a BIF implemented in C; otherwise returns false. This BIF is useful for builders of cross reference tools.

\textbf{is\_float(Term) \to bool()}

\textbf{Types:}
- Term = term()
  Returns true if Term is a floating point number; otherwise returns false.
  Allowed in guard tests.

\textbf{is\_function(Term) \to bool()}

\textbf{Types:}
- Term = term()
  Returns true if Term is a fun; otherwise returns false.
  Allowed in guard tests.

\textbf{is\_function(Term, Arity) \to bool()}

\textbf{Types:}
- Term = term()
- Arity = int()
Returns true if `Term` is a fun that can be applied with `Arity` number of arguments; otherwise returns false.
Allowed in guard tests.

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

```erlang
is_integer(Term) -> bool()
Types:
  • Term = term()
Returns true if `Term` is an integer; otherwise returns false.
Allowed in guard tests.
```

```erlang
is_list(Term) -> bool()
Types:
  • Term = term()
Returns true if `Term` is a list with zero or more elements; otherwise returns false.
Allowed in guard tests.
```

```erlang
is_number(Term) -> bool()
Types:
  • Term = term()
Returns true if `Term` is either an integer or a floating point number; otherwise returns false.
Allowed in guard tests.
```

```erlang
is_pid(Term) -> bool()
Types:
  • Term = term()
Returns true if `Term` is a pid (process identifier); otherwise returns false.
Allowed in guard tests.
```

```erlang
is_port(Term) -> bool()
Types:
  • Term = term()
Returns true if `Term` is a port identifier; otherwise returns false.
Allowed in guard tests.
```

```erlang
is_process_alive(Pid) -> bool()
```
Types:
• Pid = pid()

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

\[ \texttt{is_record}(	exttt{Term}, \texttt{RecordTag}) \rightarrow \texttt{bool}() \]

Types:
• \texttt{Term} = \texttt{term}()
• \texttt{RecordTag} = \texttt{atom}()

Returns true if \texttt{Term} is a tuple and its first element is \texttt{RecordTag}. Otherwise, returns false.

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

Allowed in guard tests, if \texttt{RecordTag} is a literal atom.

\[ \texttt{is_record}(	exttt{Term}, \texttt{RecordTag}, \texttt{Size}) \rightarrow \texttt{bool}() \]

Types:
• \texttt{Term} = \texttt{term}()
• \texttt{RecordTag} = \texttt{atom}()
• \texttt{Size} = \texttt{int}()

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

Allowed in guard tests, provided that \texttt{RecordTag} is a literal atom and \texttt{Size} is a literal integer.

**Note:**
This BIF is documented for completeness. In most cases \texttt{is_record/2} should be used.

\[ \texttt{is_reference}(	exttt{Term}) \rightarrow \texttt{bool}() \]

Types:
• \texttt{Term} = \texttt{term}()

Returns true if \texttt{Term} is a reference; otherwise returns false.

Allowed in guard tests.

\[ \texttt{is_tuple}(	exttt{Term}) \rightarrow \texttt{bool}() \]

Types:
• Term = term()
Returns true if Term is a tuple; otherwise returns false.
Allowed in guard tests.

length(List) -> int()
Types:
• List = [term()]
Returns the length of List.
> length([1,2,3,4,5,6,7,8,9]).
9
Allowed in guard tests.

link(Pid) -> true
Types:
• Pid = pid() | port()
Creates a link between the calling process and another process (or port) Pid, if there is not such a link already. If a process attempts to create a link to itself, nothing is done.
Returns true.
If Pid does not exist, the behavior of the BIF depends on if the calling process is trapping exits or not (see process_flag/2 [page 108]):
• If the calling process is not trapping exits, and checking Pid is cheap - that is, if Pid is local - link/1 fails with reason noproc.
• Otherwise, if the calling process is trapping exits, and/or Pid is remote, link/1 returns true, but an exit signal with reason noproc is sent to the calling process.

list_to_atom(String) -> atom()
Types:
• String = string()
Returns the atom whose text representation is String.
> list_to_atom("Erlang").
'Erlang'

list_to_binary(IoList) -> binary()
Types:
• IoList = iolist()
Returns a binary which is made from the integers and binaries in IoList.
list_to_binary([Bin1, 1, [2, Bin2], 4 | Bin3]).

list_to_existing_atom(String) -> atom()

Types:
  • String = string()

Returns the atom whose text representation is String, but only if there already exists such atom.
Failure: badarg if there does not already exist an atom whose text representation is String.

list_to_float(String) -> float()

Types:
  • String = string()

Returns the float whose text representation is String.
> list_to_float("2.2017764e+0").
2.20178

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

list_to_integer(String) -> int()

Types:
  • String = string()

Returns an integer whose text representation is String.
> list_to_integer("123").
123

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

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

Types:
  • String = string()
  • Base = 2..36

Returns an integer whose text representation in base Base is String.
> erlang:list_to_integer("3FF", 16).
1023
Failure: badarg if String contains a bad representation of an integer.

list_to_pid(String) -> pid()
Types:
  • String = string()
Returns a pid whose text representation is String.

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

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

Failure: badarg if String contains a bad representation of a pid.

list_to_tuple(List) -> tuple()
Types:
  • List = [term()]
Returns a tuple which corresponds to List. List can contain any Erlang terms.

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

load_module(Module, Binary) -> {module, Module} | {error, Reason}
Types:
  • Module = atom()
  • Binary = binary()
  • Reason = badfile | not_purged | badfile
If Binary contains the object code for the module Module, this BIF loads that object code. Also, if the code for the module Module already exists, all export references are replaced so they point to the newly loaded code. The previously loaded code is kept in the system as old code, as there may still be processes which are executing that code. It returns either {module, Module}, or {error, Reason} if loading fails. Reason is one of the following:

badfile The object code in Binary has an incorrect format.
not_purged Binary contains a module which cannot be loaded because old code for this module already exists.
badfile The object code contains code for another module than Module.
Warning:
This BIF is intended for the code server (see code(3) [page 45]) and should not be used elsewhere.

erlang:loaded() -> [Module]
Types:
  • Module = atom()
Returns a list of all loaded Erlang modules (current and/or old code), including preloaded modules.
See also code(3) [page 45].

erlang:localtime() -> {Date, Time}
Types:
  • Date = {Year, Month, Day}
  • Time = {Hour, Minute, Second}
  • Year = Month = Day = Hour = Minute = Second = int()
Returns the current local date and time {{Year, Month, Day}, {Hour, Minute, Second}}.
The time zone and daylight saving time correction depend on the underlying OS.
> erlang:localtime().
{{1996,11,6},{14,45,17}}

erlang:localtime_to_universaltime([Date1, Time1]) -> {Date2, Time2}
Types:
  • Date1 = Date2 = {Year, Month, Day}
  • Time1 = Time2 = {Hour, Minute, Second}
  • Year = Month = Day = Hour = Minute = Second = int()
Converts local date and time to Universal Time Coordinated (UTC), if this is supported by the underlying OS. Otherwise, no conversion is done and {Date1, Time1} is returned.
> erlang:localtime_to_universaltime([[1996,11,6],[14,45,17]]).
{{1996,11,6},{13,45,17}}

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

erlang:localtime_to_universaltime([Date1, Time1], IsDst) -> {Date2, Time2}
Types:
  • Date1 = Date2 = {Year, Month, Day}
  • Time1 = Time2 = {Hour, Minute, Second}
  • Year = Month = Day = Hour = Minute = Second = int()
  • IsDst = true | false | undefined
Converts local date and time to Universal Time Coordinated (UTC) just like erlang:localtime_to_universaltime/1, but the caller decides if daylight saving time is active or not.

If IsDst == true the \{Date1, Time1\} is during daylight saving time, if IsDst == false it is not, and if IsDst == undefined the underlying OS may guess, which is the same as calling erlang:localtime_to_universaltime([Date1, Time1]).

```erlang
> erlang:localtime_to_universaltime([[1996,11,6],[14,45,17]], true).
[[1996,11,6],[12,45,17]]
> erlang:localtime_to_universaltime([[1996,11,6],[14,45,17]], false).
[[1996,11,6],[13,45,17]]
[[1996,11,6],[13,45,17]]
```

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

\texttt{make_ref}() \rightarrow \texttt{ref()}

Returns an almost unique reference.
The returned reference will reoccur after approximately \(2^{82}\) calls; therefore it is unique enough for practical purposes.

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

erlang:make_tuple(Arity, InitialValue) \rightarrow \texttt{tuple()}

Types:
- Arity = \texttt{int()}
- InitialValue = \texttt{term()}

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

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

erlang:md5(Data) \rightarrow \texttt{Digest}

Types:
- Data = \texttt{iodata()}
- Digest = \texttt{binary()}

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

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

erlang:md5_final(Context) \rightarrow \texttt{Digest}

Types:
- Context = Digest = \texttt{binary()}

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

```erlang
erlang:md5_init() -> Context
```

Types:
- Context = binary()

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

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

Types:
- Data = iodata()
- Context = NewContext = binary()

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

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

Types:
- Type, Size – see below

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

- `total` The total amount of memory currently allocated, which is the same as the sum of memory size for processes and system.
- `processes` The total amount of memory currently allocated by the Erlang processes.
- `processes_used` The total amount of memory currently used by the Erlang processes. This memory is part of the memory presented as processes memory.
- `system` The total amount of memory currently allocated by the emulator that is not directly related to any Erlang process. Memory presented as processes is not included in this memory.
- `atom` The total amount of memory currently allocated for atoms. This memory is part of the memory presented as system memory.
- `atom_used` The total amount of memory currently used for atoms. This memory is part of the memory presented as atom memory.
- `binary` The total amount of memory currently allocated for binaries. This memory is part of the memory presented as system memory.
- `code` The total amount of memory currently allocated for Erlang code. This memory is part of the memory presented as system memory.
- `ets` The total amount of memory currently allocated for ets tables. This memory is part of the memory presented as system memory.
- `maximum` The maximum total amount of memory allocated since the emulator was started. This tuple is only present when the emulator is run with instrumentation. For information on how to run the emulator with instrumentation see `[instrument(3)]` and/or `[erl(1)]`.
Note:
The `system` value is not complete. Some allocated memory that should be part of the `system` value are not. For example, memory allocated by drivers is missing.

When the emulator is run with instrumentation, the `system` value is more accurate, but memory directly allocated by `malloc` (and friends) are still not part of the `system` value. Direct calls to `malloc` are only done from OS specific runtime libraries and perhaps from user implemented Erlang drivers that do not use the memory allocation functions in the driver interface.

Since the `total` value is the sum of `processes` and `system` the error in `system` will propagate to the `total` value.

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

\[
\begin{align*}
total &= \text{processes} + \text{system} \\
\text{processes} &= \text{processes\_used} + \text{Processes\_Not\_Used} \\
\text{system} &= \text{atom} + \text{binary} + \text{code} + \text{ets} + \text{Other\_System} \\
\text{atom} &= \text{atom\_used} + \text{Atom\_Not\_Used} \\
\end{align*}
\]

\[\text{Real\_Total} = \text{processes} + \text{Real\_System}\]
\[\text{Real\_System} = \text{system} + \text{Missed\_System}\]

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

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

```
derlang:memory(Type | [Type]) -> Size | [{Type, Size}]
Types:
  • Type, Size - see below
```

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

See `erlang:memory/0` [page 97].

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

```
module_loaded(Module) -> bool()
```

Types:
Kernel Reference Manual

- Module = atom()
  Returns true if the module Module is loaded, otherwise returns false. It does not attempt to load the module.

**Warning:**
This BIF is intended for the code server (see code(3) [page 45]) and should not be used elsewhere.

erlang:monitor(Type, Item) -> MonitorRef

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

The calling process starts monitoring Item which is an object of type Type.
Currently only processes can be monitored, i.e. the only allowed Type is process, but other types may be allowed in the future.

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

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

{"DOWN", MonitorRef, Type, Object, Info}

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

**Object** A reference to the monitored object:
- the pid of the monitored process, if Item was specified as a pid.
- {RegName, Node}, if Item was specified as {RegName, Node}.
- {RegName, Node}, if Item was specified as RegName. Node will in this case be the name of the local node (node()).
Info Either the exit reason of the process, \texttt{noproc} (non-existing process), or \texttt{noconnection} (no connection to Node).

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

The monitoring is turned off either when the \texttt{DOWN} message is sent, or when \texttt{erlang:demonitor/1} [page 78] is called.

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

Making several calls to \texttt{erlang:monitor/2} for the same \texttt{Item} is not an error; it results in as many, completely independent, monitorings.

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

\begin{verbatim}
monitor_node(Node, Flag) -> true
Types:
  * Node = node()
  * Flag = bool()

Monitors the status of the node \texttt{Node}. If \texttt{Flag} is \texttt{true}, monitoring is turned on; if \texttt{Flag} is \texttt{false}, monitoring is turned off.

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

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

Nodes connected through hidden connections can be monitored as any other node.

Failure: \texttt{badarg} if the local node is not alive.
\end{verbatim}
Types:
- `Node = node()`

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

Allowed in guard tests.

`node(Arg) -> Node`

Types:
- `Arg = pid() | port() | ref()`
- `Node = node()`

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

Allowed in guard tests.

`nodes() -> Nodes`

Types:
- `Nodes = [node()]`

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

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

Types:
- `Arg = visible | hidden | connected | this | known`
- `Nodes = [node()]`

Returns a list of nodes according to argument given. The result returned when the argument is a list, is the list of nodes satisfying the disjunction(s) of the list elements. `Arg` can be any of the following:

- `visible` Nodes connected to this node through normal connections.
- `hidden` Nodes connected to this node through hidden connections.
- `connected` All nodes connected to this node.
- `this` This node.
- `known` Nodes which are known to this node, i.e., connected, previously connected, etc.

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

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

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

Types:
- `MegaSecs = Secs = MicroSecs = int()`
Returns the tuple \{MegaSecs, Secs, MicroSecs\} which is the elapsed time since 
00:00 GMT, January 1, 1970 (zero hour) on the assumption that the underlying OS 
supports this. Otherwise, some other point in time is chosen. It is also guaranteed that 
subsequent calls to this BIF returns continuously increasing values. Hence, the return 
value from \texttt{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.

\begin{verbatim}
open_port(PortName, PortSettings) -> port()

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

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

\begin{verbatim}
\{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 \texttt{vfork} is used in 
preference to \texttt{fork} for performance reasons, although it has a history of being less 
robust. If there are problems with using \texttt{vfork}, setting the environment variable 
\texttt{ERL\_NO\_VFORK} to any value will cause \texttt{fork} to be used instead.
\{fd, In, Out\} Allows an Erlang process to access any currently opened file descriptors 
used by Erlang. The file descriptor In can be used for standard input, and the file 
descryptor 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.
\end{verbatim}

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

\begin{verbatim}
\{packet, N\} Messages are preceded by their length, sent in \texttt{N} bytes, with the most 
significant byte first. Valid values for \texttt{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.
\end{verbatim}

\end{verbatim}
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 \( \text{Flag, Line} \), where \( \text{Flag} \) is either \( \text{eol} \) or \( \text{noeol} \) and \( \text{Line} \) is the actual data delivered (without the newline sequence).

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

The \{packet, \( N \)\} and \{line, \( L \)\} settings are mutually exclusive.

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

\( \text{env, Env} \) This is only valid for \{spawn, \text{Command}\}. The environment of the started process is extended using the environment specifications in \( \text{Env} \).

\( \text{Env} \) should be a list of tuples \{\text{Name, Val}\}, where \( \text{Name} \) is the name of an environment variable, and \( \text{Val} \) is the value it is to have in the spawned port process. Both \( \text{Name} \) and \( \text{Val} \) must be strings. The one exception is \( \text{Val} \) being the atom \text{false} (in analogy with \text{os:getenv/1}), which removes the environment variable. Not available on VxWorks.

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

When the external process connected to the port exits, a message of the form \{\text{Port, \text{exit_status, Status}}\} is sent to the connected process, where \( \text{Status} \) 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., \( 128+\text{signal} \)).

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

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

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

\text{nouse_stdio} \) The opposite of \text{use_stdio}. Uses file descriptors 3 and 4 for communication with Erlang.

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

\text{in} \) The port can only be used for input.

\text{out} \) The port can only be used for output.

\text{binary} \) All IO from the port are binary data objects as opposed to lists of bytes.

\text{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 \{\text{Port, eof}\} message will be sent to the process holding the port.

The default is \text{stream} for all types of port and \text{use_stdio} for spawned ports.

Failure: If the port cannot be opened, the exit reason is the Posix error code which most closely describes the error, or \text{einval} if no Posix code is appropriate. The following Posix error codes may appear:
erlang Kernel Reference Manual

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.
emfile A file or port table is full.

During use of a port opened using \{\texttt{spawn}, Name\}, errors arising when sending
messages to it are reported to the owning process using signals of the form \{\texttt{\textsc{EXIT}},
Port, PosixCode\}. See file(3) for possible values of PosixCode.

The maximum number of ports that can be open at the same time is 1024 by default,
but can be configured by the environment variable \texttt{ERL\_MAX\_PORTS}.

erlang:phash(Term, Range) \rightarrow Hash

Types:
- \texttt{Term = term()}
- \texttt{Range = 1..2^{32}}
- \texttt{Hash = 1..Range}

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

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

erlang:phash2(Term [, Range]) \rightarrow Hash

Types:
- \texttt{Term = term()}
- \texttt{Range = 1..2^{32}}
- \texttt{Hash = 0..Range-1}

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

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

Note that the range \(0..Range-1\) is different from the range of \texttt{phash/2 (1..Range)}.

pid_to_list(Pid) \rightarrow string()

Types:
- \texttt{Pid = pid()}
Returns a string which corresponds to the text representation of Pid.

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

```
pport_close(Port) -> true

Types:
  • Port = port() | atom()

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

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

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

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

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

```
pport_command(Port, Data) -> true

Types:
  • Port = port() | atom()
  • Data = iodata()

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

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

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

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

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

```
pport_connect(Port, Pid) -> true
```

Kernel Application (KERNEL)
Types:
- Port = port() | atom()
- Pid = pid()

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

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

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

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

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

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

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

port_control(Port, Operation, Data) -> Res

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

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

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

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

erlang:port_call(Port, Operation, Data) -> term()

Types:
- Port = port() | atom()
- Operation = int()
- Data = term()
Performs a synchronous call to a port. The meaning of Operation and Data depends on the port, i.e., on the port driver. Not all port drivers support this feature.

Port is a port identifier, referring to a driver.

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

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

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

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

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

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

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

- {registered_name, RegName} RegName (an atom) is the registered name of the port. If the port has no registered name, this tuple is not present in the list.
- {id, Index} Index (an integer) is the internal index of the port. This index may be used to separate ports.
- {connected, Pid} Pid is the process connected to the port.
- {links, Pids} Pids is a list of pids to which processes the port is linked.
- {name, String} String is the command name set by open_port.
- {input, Bytes} Bytes is the total number of bytes read from the port.
- {output, Bytes} Bytes is the total number of bytes written to the port.

Failure: badarg if Port is not a local port.

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

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

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

For valid values of Item, and corresponding values of Info, see erlang:port_info/1 [page 107].

Failure: badarg if Port is not a local port.

```
erlang:port_to_list(Port) -> string()
```

Types:
Port = port()

Returns a string which corresponds to the text representation of the port identifier Port.

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

erlang:ports() -> [port()]

Returns a list of all ports on the local node.

pre_loaded() -> [Module]

Types:
- Module = atom()

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

erlang:process_display(Pid, Type) -> void()

Types:
- Pid = pid()
- Type = backtrace

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

process_flag(Flag, Value) -> OldValue

Types:
- Flag, Value, OldValue - see below

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

process_flag(trap_exit, Boolean) When `trap_exit` is set to true, exit signals arriving to a process are converted to `$EXIT$, From, Reason$ messages, which can be received as ordinary messages. If `trap_exit` is set to false, the process exits if it receives an exit signal other than `normal` and the exit signal is propagated to its linked processes. Application processes should normally not trap exits. See also `exit/2` [page 81].

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

process_flag(min_heap_size, MinHeapSize) This changes the minimum heap size for the calling process.
**process_flag(priority, Level)** This sets the process priority. Level is an atom. All implementations support three priority levels, low, normal, and high. The default is normal.

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

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

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

Sets certain flags for the process Pid, in the same manner as **process_flag/2 [page 108]**. Returns the old value of the flag. The allowed values for Flag are only a subset of those allowed in **process_flag/2**, namely: save_calls.

Failure: badarg if Pid is not a local process.

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

Types:
- Pid = pid()
- Item, Info - see below

Returns a list containing tuples with information about the process Pid, or undefined if the process is not alive. The order of the tuples is not defined, nor are all the tuples mandatory.

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

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

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

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

{group_leader, GroupLeader} GroupLeader is group leader for the IO of the process.

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

{initial_call, {Module, Function, Arity}} Module, Function, Arity is the initial function call with which the process was spawned.
{links, Pids} Pids is a list of pids, with processes to which the process has a link.
{message_queue_len, MessageQueueLen} MessageQueueLen is the number of messages currently in the message queue of the process. This is the length of the list MessageQueue returned as the info item messages (see below).
{messages, MessageQueue} MessageQueue is a list of the messages to the process which have not yet been processed.
{priority, Level} Level is the current priority level for the process. Only low and normal are always supported.
{reductions, Number} Number is the number of reductions executed by the process.
{registered_name, Atom} Atom is the registered name of the process. If the process has no registered name, this tuple is not present in the list.
{stack_size, Size} Size is the stack size of the process in words.
{status, Status} Status is the status of the process. Status is waiting (waiting for a message), running, runnable (ready to run, but another process is running), or suspended (suspended on a “busy” port or by the erlang:suspend_process/1 BIF).
{trap_exit, Boolean} Boolean is true if the process is trapping exits, otherwise it is false.

Failure: badarg if Pid is not a local process.

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

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

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

The value of Item, and corresponding value of Info, can be any of the values specified for process_info/1 [page 109].

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

{backtrace, Bin} The binary Bin contains the same information as the output from erlang:process_display(Pid, backtrace). Use binary_to_list/1 to obtain the string of characters from the binary.
{last_calls, false|Calls} The value is false if call saving is not active for the process (see process_flag/3 [page 109]). If call saving is active, a list is returned, in which the last element is the most recent called.
{memory, Size} Size is the size of the process in bytes. This includes call stack, heap and internal structures.
{monitored_by, Pids} A list of pids that are monitoring the process (with erlang:monitor/2).
{monitors, Monitors} A list of monitors (started by erlang:monitor/2) that are active for the process. For a local process monitor or a remote process monitor by pid, the list item is {process, Pid}, and for a remote process monitor by name, the list item is {process, {RegName, Node}}.
Note however, that not all implementations support every one of the above Items.
Failure: badarg if Pid is not a local process.

processes() -> [pid()]
Returns a list of all processes on the local node.
> processes().
[<0.0.0>,
 <0.2.0>,
 <0.4.0>,
 <0.5.0>,
 <0.7.0>,
 <0.8.0>]

purge_module(Module) -> void()
Types:
- Module = atom()
Removes old code for Module. Before this BIF is used, erlang:check_process_code/2
should be called to check that no processes are executing old code in the module.

Warning:
This BIF is intended for the code server (see code(3) [page 45]) and should not be
used elsewhere.

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

put(Key, Val) -> OldVal | undefined
Types:
- Key = Val = OldVal = term()
Adds a new Key to the process dictionary, associated with the value Val, and returns
undefined. If Key already exists, the old value is deleted and replaced by Val and the
function returns the old value.

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

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

erlang:raise(Class, Reason, Stacktrace)
Types:
- Class = error | exit | throw
- Reason = term()
- Stacktrace = [[Module, Function, Arity | Args] | [Fun, Args]]
- Module = Function = atom()
- Arity = int()
- Args = [term()]
- Fun = [fun()]

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

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

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

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

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

```
erlang:read_timer(TimerRef) -> int() | false
```

Types:
- TimerRef = ref()

TimerRef is a timer reference returned by erlang:send_after/3 [page 115] or
erlang:start_timer/3 [page 121]. If the timer is active, the function returns the time in
milliseconds left until the timer will expire, otherwise false (which means that
TimerRef was never a timer, that it has been cancelled, or that it has already delivered
its message).

See also erlang:send_after/3 [page 115], erlang:start_timer/3 [page 121], and
erlang:cancel_timer/1 [page 76].

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

Types:
- Ref = ref()
Returns a string which corresponds to the text representation of Ref.

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

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

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

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

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

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

```erlang
registered() -> [RegName]

Types:
- RegName = atom()

Returns a list of names which have been registered using register/2 [page 113].

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

```erlang
erlang:resume_process(Pid) -> true

Types:
- Pid = pid()

Resume a suspended process Pid.

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

Failure: badarg if Pid does not exist.

```erlang
round(Number) -> int()

Types:
```
Number = number()
Returns an integer by rounding Number.
> \texttt{round(5.5)}.
6
Allowed in guard tests.

\texttt{self()} \rightarrow \texttt{pid()}
Returns the pid (process identifier) of the calling process.
> \texttt{self().}
\texttt{<0.26.0>}
Allowed in guard tests.

erlang:send(Dest, Msg) \rightarrow \texttt{Msg} \newline
Types:
\begin{itemize}
\item Dest = \texttt{pid()} \mid \texttt{port()} \mid \texttt{RegName} \mid \{\texttt{RegName}, \texttt{Node}\}
\item Msg = \texttt{term()}
\item RegName = \texttt{atom()}
\item Node = \texttt{node()}
\end{itemize}
Sends a message and returns \texttt{Msg}. This is the same as \texttt{Dest ! Msg}.
\texttt{Dest} may be a remote or local pid, a (local) port, a locally registered name, or a tuple \{\texttt{RegName}, \texttt{Node}\} for a registered name at another node.

erlang:send(Dest, Msg, [Option]) \rightarrow \texttt{Res} \newline
Types:
\begin{itemize}
\item Dest = \texttt{pid()} \mid \texttt{port()} \mid \texttt{RegName} \mid \{\texttt{RegName}, \texttt{Node}\}
\item RegName = \texttt{atom()}
\item Node = \texttt{node()}
\item Msg = \texttt{term()}
\item Option = \texttt{nosuspend} \mid \texttt{noconnect}
\item Res = \texttt{ok} \mid \texttt{nosuspend} \mid \texttt{noconnect}
\end{itemize}
Sends a message and returns \texttt{ok}, or does not send the message but returns something else (see below). Otherwise the same as \texttt{erlang:send/2 [page 114]}. See also \texttt{erlang:send_nosuspend/2,3 [page 115]} for more detailed explanation and warnings.
The possible options are:
\begin{itemize}
\item \texttt{nosuspend} If the sender would have to be suspended to do the send, \texttt{nosuspend} is returned instead.
\item \texttt{noconnect} If the destination node would have to be autoconnected before doing the send, \texttt{noconnect} is returned instead.
\end{itemize}
**Warning:**
As with erlang:send_nosuspend/2,3: Use with extreme care!

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

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

Starts a timer which will send the message Msg to Dest after Time milliseconds.
If Dest is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process.

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

See also erlang:start_timer/3 [page 121], erlang:cancel_timer/1 [page 76], and erlang:read_timer/1 [page 112].

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

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

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

The same as erlang:send(Dest, Msg, [nosuspend]) [page 114], but returns true if the message was sent and false if the message was not sent because the sender would have had to be suspended.

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

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

This function is only to be used in very rare circumstances where a process communicates with Erlang nodes that can disappear without any trace causing the TCP buffers and the drivers queue to be overfull before the node will actually be shut down (due to tick timeouts) by net_kernel. The normal reaction to take when this happens is some kind of premature shutdown of the other node.
Note that ignoring the return value from this function would result in unreliable message passing, which is contradictory to the Erlang programming model. The message is not sent if this function returns false.

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

**Warning:**
Use with extreme care!

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

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

The same as erlang:send(Dest, Msg, [nosuspend | Options]) [page 114], but with boolean return value.

This function behaves like erlang:send_nosuspend/2 [page 115], but takes a third parameter, a list of options. The only currently implemented option is noconnect. The option noconnect makes the function return false if the remote node is not currently reachable by the local node. The normal behaviour is to try to connect to the node, which may stall the process for a shorter period. The use of the noconnect option makes it possible to be absolutely sure not to get even the slightest delay when sending to a remote process. This is especially useful when communicating with nodes who expect to always be the connecting part (i.e. nodes written in C or Java).

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

**Warning:**
Use with extreme care!

erlang:set_cookie(Node, Cookie) -> true

Types:
- Node = node()
- Cookie = atom()
Sets the magic cookie of Node to the atom Cookie. If Node is the local node, the function also sets the cookie of all other unknown nodes to Cookie (see [Distributed Erlang] in the Erlang Reference Manual).
Failure: function clause if the local node is not alive.

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

Types:
- Index = 1..size(Tuple1)
- Tuple1 = Tuple2 = tuple()
- Value = term()
Returns a tuple which is a copy of the argument Tuple1 with the element given by the integer argument Index (the first element is the element with index 1) replaced by the argument Value.

```erlang
> setelement(2, [10, green, bottles], red).
[10, red, bottles]
```

**size(Item) -> int()**

Types:
- Item = tuple() \| binary()
Returns an integer which is the size of the argument Item, which must be either a tuple or a binary.

```erlang
> size([morni, mulle, bwange]).
3
```

Allowed in guard tests.

**spawn(Fun) -> pid()**

Types:
- Fun = fun()
Returns the pid of a new process started by the application of Fun to the empty list []. Otherwise works like spawn/3 [page 117].

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

Types:
- Node = node()
- Fun = fun()
Returns the pid of a new process started by the application of Fun to the empty list [] on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn/3 [page 117].

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

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

error_handler:undefined_function(Module, Function, Args) is evaluated by the new process if Module:Function/Arity does not exist (where Arity is the length of Args). The error handler can be redefined (see process_flag/2 [page 108]). If error_handler is undefined, or the user has redefined the default error_handler its replacement is undefined, a failure with the reason undef will occur.

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

\[\text{spawn(Node, Module, Function, ArgumentList) \rightarrow pid()}\]

Types:
- \(\text{Node = node()}\)
- \(\text{Module = Function = atom()}\)
- \(\text{Args = [term()]}\)

Returns the pid of a new process started by the application of Module:Function to Args on Node. If Node does not exists, a useless pid is returned. Otherwise works like spawn/3 [page 117].

\[\text{spawn_link(Func) \rightarrow pid()}\]

Types:
- \(\text{Func = fun()}\)

Returns the pid of a new process started by the application of Func to the empty list []. A link is created between the calling process and and the new process, atomically. Otherwise works like spawn/3 [page 117].

\[\text{spawn_link(Node, Func) \rightarrow pid()}\]

Types:
- \(\text{Node = node()}\)
- \(\text{Func = fun()}\)

Returns the pid of a new process started by the application of Func to the empty list [] on Node. A link is created between the calling process and and the new process, atomically. If Node does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason noconnection will be received). Otherwise works like spawn/3 [page 117].

\[\text{spawn_link(Module, Function, Args) \rightarrow pid()}\]

Types:
- \(\text{Module = Function = atom()}\)
- \(\text{Args = [term()]}\)
Returns the pid of a new process started by the application of Module:Function to Args. A link is created between the calling process and the new process, atomically. Otherwise works like spawn/3 [page 117].

\[
\text{spawn_link(Node, Module, Function, Args) -> pid()}
\]

Types:
- \( \text{Node} = \text{node()} \)
- \( \text{Module} = \text{Function} = \text{atom()} \)
- \( \text{Args} = [\text{term()}] \)

Returns the pid of a new process started by the application of Module:Function to Args on Node. A link is created between the calling process and the new process, atomically. If Node does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason noconnection will be received). Otherwise works like spawn/3 [page 117].

\[
\text{spawn_opt(Fun, [Option]) -> pid()}
\]

Types:
- \( \text{Fun} = \text{fun()} \)
- \( \text{Option} = \text{link} \mid \{\text{priority, Level}\} \mid \{\text{fullsweep_after, Number}\} \mid \{\text{min_heap_size, Size}\} \)
- \( \text{Level} = \text{low} \mid \text{normal} \mid \text{high} \)
- \( \text{Number} = \text{int()} \)
- \( \text{Size} = \text{int()} \)

Returns the pid of a new process started by the application of Fun to the empty list \([]\). Otherwise works like spawn_opt/4 [page 119].

\[
\text{spawn_opt(Node, Fun, [Option]) -> pid()}
\]

Types:
- \( \text{Node} = \text{node()} \)
- \( \text{Fun} = \text{fun()} \)
- \( \text{Option} = \text{link} \mid \{\text{priority, Level}\} \mid \{\text{fullsweep_after, Number}\} \mid \{\text{min_heap_size, Size}\} \)
- \( \text{Level} = \text{low} \mid \text{normal} \mid \text{high} \)
- \( \text{Number} = \text{int()} \)
- \( \text{Size} = \text{int()} \)

Returns the pid of a new process started by the application of Fun to the empty list \([]\) on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn_opt/4 [page 119].

\[
\text{spawn_opt(Module, Function, Args, [Option]) -> pid()}
\]

Types:
- \( \text{Module} = \text{Function} = \text{atom()} \)
- \( \text{Args} = [\text{term()}] \)
- \( \text{Option} = \text{link} \mid \{\text{priority, Level}\} \mid \{\text{fullsweep_after, Number}\} \mid \{\text{min_heap_size, Size}\} \)
- \( \text{Level} = \text{low} \mid \text{normal} \mid \text{high} \)
- Number = int()
- Size = int()

Works exactly like spawn/3 [page 117], except that an extra option list is 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. The fullsweep_after option makes it possible to specify the maximum number of generational collections before forcing a fullsweep even if there is still room on the old heap. Setting the number to zero effectively disables the general collection algorithm, meaning that all live data is copied at every garbage collection. Here are a few cases when it could be useful to change fullsweep_after. Firstly, if binaries that are no longer used should be thrown away as soon as possible. (Set Number to zero.) Secondly, a process that mostly have short-lived data will be fullsweeped seldom or never, meaning that the old heap will contain mostly garbage. To ensure a fullsweep once in a while, set Number to a suitable value such as 10 or 20. Thirdly, in embedded systems with limited amount of RAM and no virtual memory, one might want to preserve memory by setting Number to zero. (The value may be set globally, see erlang:system_flag/2 [page 123].)

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

**spawn_opt**

spawn_opt(Node, Module, Function, Args, [Option]) -> pid()

Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]
- Option = link | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size} | Level = low | normal | high
Kernel Reference Manual

- Number = int()
- Size = int()

Returns the pid of a new process started by the application of Module:Function to Args on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn_opt/4 [page 119].

split_binary(Bin, Pos) -> {Bin1, Bin2}

Types:
  - Bin = Bin1 = Bin2 = binary()
  - Pos = 1..size(Bin)

Returns a tuple containing the binaries which are the result of splitting Bin into two parts at position Pos. This is not a destructive operation. After the operation, there will be three binaries altogether.

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

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

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

Starts a timer which will send the message {timeout, TimerRef, Msg} to Dest after Time milliseconds.

If Dest is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process.

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

See also erlang:send_after/3 [page 115], erlang:cancel_timer/1 [page 76], and erlang:read_timer/1 [page 112].

Failure: badarg if the arguments does not satisfy the requirements specified above.
statistics(Type) -> Res

Types:
- Type, Res - see below

Returns information about the system as specified by Type:

context_switches Returns {ContextSwitches, 0}, where ContextSwitches is the total number of context switches since the system started.

exact_reductions Returns {Total_Exact_Reductions, Exact_Reductions_Since_Last_Call}.

NOTE: statistics(exact_reductions) is a more expensive operation than statistics(reductions) [page 122] especially on an Erlang machine with SMP support.

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

io Returns {{input, Input}, {output, Output}}, where Input is the total number of bytes received through ports, and Output is the total number of bytes output to ports.

reductions Returns {Total_Reductions, Reductions_Since_Last_Call}.

NOTE: From erts version 5.5 (OTP release R11B) this value does not include reductions performed in current time slices of currently scheduled processes. If an exact value is wanted, use statistics(exact_reductions) [page 122].

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

runtime Returns {Total_Run_Time, Time_Since_Last_Call}.

wall_clock Returns {Total_Wallclock_Time, Wallclock_Time_Since_Last_Call}.

wall_clock can be used in the same manner as runtime, except that real time is measured as opposed to runtime or CPU time.

All times are in milliseconds.

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

erlang:suspend_process(Pid) -> true

Types:
- Pid = pid()

Suspends the process Pid.

Warning:
This BIF is intended for debugging only.

Failure: badarg if Pid does not exist.
erlang:system_flag(Flag, Value) -> OldValue

Types:
- Flag, Value, OldValue - see below

Sets various system properties of the Erlang node. Returns the old value of the flag.

erlang:system_flag(backtrace_depth, Depth) Sets the maximum depth of call stack backtraces in the exit reason element of 'EXIT' tuples.

erlang:system_flag(fullsweep_after, Number) Number is a non-negative integer which indicates how many times generational garbages collections can be done without forcing a fullsweep collection. The value applies to new processes; processes already running are not affected.

In low-memory systems (especially without virtual memory), setting the value to 0 can help to conserve memory.

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

erlang:system_flag(min_heap_size, MinHeapSize) Sets the default minimum heap size for processes. The size is given in words. The new min_heap_size only effects processes spawned after the change of min_heap_size has been made. The min_heap_size can be set for individual processes by use of spawn_opt/N [page 119] or process_flag/2 [page 108].

erlang:system_flag(schedulers, NO) Sets the number of scheduler threads to used when the runtime system has been compiled with smp support. Valid range of NO is 1-1024. Returns {OpRes, NoOfSchedulers, OldNoOfSchedulers} where OpRes = atom() (ok or a posix error), NoOfSchedulers = integer() (current number of scheduler threads), and OldNoOfSchedulers = integer() (previous number of scheduler threads).

NOTE! This feature might be removed without prior notice.

NOTE! This is a quite expensive operation. Do not use it for mutual exclusion.

erlang:system_flag(trace_control_word, TCW) Sets the value of the node's trace control word to TCW. TCW should be an unsigned integer. For more information see documentation of the [set_tcw] function in the match specification documentation in the ERTS User's Guide.

erlang:system_info(Type) -> Res

Types:
- Type, Res - see below

Returns various information about the current system (emulator) as specified by Type:

allocated_areas Returns a list of tuples with information about miscellaneous allocated memory areas.

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

erlang:system_info(allocated_areas) is intended for debugging, and the content is highly implementation dependent. The content of the results will therefore change when needed without prior notice.
Note: The sum of these values is not the total amount of memory allocated by the emulator. Some values are part of other values, and some memory areas are not part of the result. If you are interested in the total amount of memory allocated by the emulator see erlang:memory/0,1 [page 97].

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

Types:
- Allocator = undefined | elib_malloc | glibc
- Version = [int()]
- Features = [atom()]
- Settings = [{Subsystem, [{Parameter, Value}]},  
  Subsystem = atom(),  
  Parameter = atom(),  
  Value = term()]

Explanation:
- Allocator corresponds to the malloc() implementation used. If Allocator equals undefined, the malloc() implementation used could not be identified. Currently elib_malloc and glibc can be identified.
- Version is a list of integers (but not a string) representing the version of the malloc() implementation used.
- Features is a list of atoms representing allocation features used.
- Settings is a list of subsystems, their configurable parameters, and used values. Settings may differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes.

See also "System Flags Effecting erts_alloc" in [erts_alloc(3)].

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

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

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

compat_rel. Returns the compatibility mode of the local node as an integer. The integer returned represents the Erlang/OTP release which the current emulator has been set to be backward compatible with. The compatibility mode can be configured at startup by using the command line flag +R, see [erl(1)].
creation Returns the creation of the local node as an integer. The creation is changed when a node is restarted. The creation of a node is stored in process identifiers, port identifiers, and references. This makes it (to some extent) possible to distinguish between identifiers from different incarnations of a node. Currently valid creations are integers in the range 1..3, but this may (probably will) change in the future. If the node is not alive, 0 is returned.

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

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

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

\{heap_size, Size\} Where \(\text{Size}\) is the current heap size.

\{max_allocated_size, Size\} Where \(\text{Size}\) is the maximum amount of memory allocated on the heap since the emulator started.

\{allocated_size, Size\} Where \(\text{Size}\) is the current amount of memory allocated on the heap.

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

\{no_allocated_blocks, No\} Where \(\text{No}\) is the current number of allocated blocks on the heap.

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

\{smallest_allocated_block, Size\} Where \(\text{Size}\) is the size of the smallest allocated block on the heap.

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

fullsweep_after Returns \{fullsweep_after, int()\} which is the fullsweep_after garbage collection setting used by default. For more information see garbage_collection described below.

garbage_collection Returns a list describing the default garbage collection settings. A process spawned on the local node by a spawn or spawn_link will use these garbage collection settings. The default settings can be changed by use of system_flag/2 [page 123]. spawn_opt/4 [page 119] can spawn a process that does not use the default settings.

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

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

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

private Each process has a heap reserved for its use and no references between heaps of different processes are allowed. Messages passed between processes are copied between heaps.
shared  One heap for use by all processes. Messages passed between processes are
   passed by reference.
hybrid  A hybrid of the private and shared heap types. A shared heap as well as
   private heaps are used.
info      Returns a binary containing a string of miscellaneous system information
   formatted as in Erlang crash dumps. For more information see the ["How to
   interpret the Erlang crash dumps"] chapter in the ERTS User's Guide.
loaded    Returns a binary containing a string of loaded module information formatted as
   in Erlang crash dumps. For more information see the ["How to interpret the Erlang
   crash dumps"] chapter in the ERTS User's Guide.
machine   Returns a string containing the Erlang machine name.
process_count  Returns the number of processes currently existing at the local node as
   an integer. The same value as length(processes()) returns.
process_limit  Returns the maximum number of concurrently existing processes at
   the local node as an integer. This limit can be configured at startup by using the
   command line flag +P, see [erl(1)].
procs        Returns a binary containing a string of process and port information formatted as
   in Erlang crash dumps. For more information see the ["How to interpret the Erlang
   crash dumps"] chapter in the ERTS User's Guide.
scheduler   Returns the scheduler id (a positive integer) of current scheduler thread.
schedulers  Returns the number of scheduler threads.
smp_support Returns true if the emulator has been compiled with smp support;
   otherwise, false is returned.
system_version Returns a string containing the emulator type and version as well as
   some important properties such as the size of the thread pool, etc.
system_architecture Returns a string containing the processor and OS architecture
   the emulator is built for.
threads     Returns true if the emulator has been compiled with thread support;
   otherwise, false is returned.
thread_pool_size Returns the number of threads used for driver calls as an integer.
trace_control_word Returns the value of the node's trace control word. For more
   information see documentation of the function get_tcw in "Match Specifications in
version     Returns a string containing the version number of the emulator.
wordsize    Returns the word size in bytes as an integer, i.e. on a 32-bit architecture 4 is
   returned, and on a 64-bit architecture 8 is returned.

erlang:system_monitor() -> MonSettings

Types:
   • MonSettings -> {MonitorPid, Options} | undefined
   • MonitorPid = pid()
   • Options = [Option]
   • Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
   • Time = Size = int()
Returns the current system monitoring settings set by \texttt{erlang:system\_monitor/2} \cite{page 127} as \{MonitorPid, Options\}, or \texttt{undefined} if there are no settings. The order of the options may be different from the one that was set.

\texttt{erlang:system\_monitor(\texttt{undefined} | \{MonitorPid, Options\}) -> MonSettings}

\textbf{Types:}

\begin{itemize}
  \item MonitorPid, Options, MonSettings – see below
\end{itemize}

When called with the argument \texttt{undefined}, all system performance monitoring settings are cleared.

Calling the function with \{MonitorPid, Options\} as argument, is the same as calling \texttt{erlang:system\_monitor(MonitorPid, Options)} \cite{page 127}.

Returns the previous system monitor settings just like \texttt{erlang:system\_monitor/0} \cite{page 126}.

\texttt{erlang:system\_monitor(MonitorPid, \[Option\]) -> MonSettings}

\textbf{Types:}

\begin{itemize}
  \item MonitorPid = \texttt{pid()}
  \item Option = \{\texttt{long\_gc, Time}\} | \{\texttt{large\_heap, Size}\} | \texttt{busy\_port} | \texttt{busy\_dist\_port}
  \item Time = Size = \texttt{int()}
  \item MonSettings = \{OldMonitorPid, \[Option\]}\n  \item OldMonitorPid = \texttt{pid()}
\end{itemize}

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

\{\texttt{long\_gc, Time}\} If a garbage collection in the system takes at least Time wallclock milliseconds, a message \{monitor, GcPid, long\_gc, Info\} is sent to MonitorPid. GcPid is the pid that was garbage collected and Info is a list of two-element tuples describing the result of the garbage collection. One of the tuples is \{timeout, GcTime\} where GcTime is the actual time for the garbage collection in milliseconds. The other are the tuples tagged with heap\_size, stack\_size, mbuf\_size and heap\_block\_size from the gc\_start trace message (see \texttt{erlang:trace/3} \cite{page 129}).

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

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

\texttt{busy\_dist\_port} If a process in the system gets suspended because it sends to a process on a remote node whose inter-node communication was handled by a busy port, a message \{monitor, SusPid, busy\_dist\_port, Port\} is sent to MonitorPid. SusPid is the pid that got suspended when sending through the inter-node communication port Port.
Returns the previous system monitor settings just like erlang:system_monitor/0 [page 126].

**Note:**
If a monitoring process gets so large that it itself starts to cause system monitor messages when garbage collecting, the messages will enlargen the process's message queue and probably make the problem worse.

Keep the monitoring process neat and do not set the system monitor limits too tight.

Failure: badarg if MonitorPid does not exist.

term_to_binary(Term) -> ext_binary()

Types:
- Term = term()

Returns a binary data object which is the result of encoding Term according to the Erlang external term format.

This can be used for a variety of purposes, for example writing a term to a file in an efficient way, or sending an Erlang term to some type of communications channel not supported by distributed Erlang.

See also binary_to_term/1 [page 76].

term_to_binary(Term, [Option]) -> ext_binary()

Types:
- Term = term()
- Option = compressed

Returns a binary data object which is the result of encoding Term according to the Erlang external term format.

If the option compressed is provided, the external term format will be compressed. The compressed format is automatically recognized by binary_to_term/1 in R7.

See also binary_to_term/1 [page 76].

throw(Any)

Types:
- Any = term()

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

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

Failure: nocatch if not evaluated within a catch.

time() -> {Hour, Minute, Second}

Types:
• \texttt{Hour = Minute = Second = int()}

Returns the current time as \texttt{\{Hour, Minute, Second\}}.

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

\begin{verbatim}
> time().
\end{verbatim}
\begin{verbatim}
\{9, 42, 44\}
\end{verbatim}

\texttt{tl(List1) \rightarrow List2}

Types:
\begin{itemize}
\item \texttt{List1} = \texttt{List2} = \texttt{[term()]}\end{itemize}

Returns the tail of \texttt{List1}, that is, the list minus the first element.

\begin{verbatim}
> tl([geesties, guilies, beasties]).
\end{verbatim}
\begin{verbatim}
\[guilies, beasties\]
\end{verbatim}

Allowed in guard tests.

Failure: \texttt{badarg} if \texttt{List} is the empty list \texttt{[]}.

\texttt{erlang:trace(PidSpec, How, FlagList) \rightarrow int()}

Types:
\begin{itemize}
\item \texttt{PidSpec} = \texttt{pid()} | \texttt{existing} | \texttt{new} | \texttt{all}
\item \texttt{How} = \texttt{bool()}
\item \texttt{FlagList} = \texttt{[Flag]}
\item \texttt{Flag} – see below\end{itemize}

Turns on (if \texttt{How} == true) or off (if \texttt{How} == false) the trace flags in \texttt{FlagList} for the process or processes represented by \texttt{PidSpec}.

\texttt{PidSpec} is either a \texttt{pid} for a local process, or one of the following atoms:
\begin{itemize}
\item \texttt{existing} All processes currently existing.
\item \texttt{new} All processes that will be created in the future.
\item \texttt{all} All currently existing processes and all processes that will be created in the future.\end{itemize}

\texttt{FlagList} can contain any number of the following flags (the “message tags” refers to the list of messages following below):
\begin{itemize}
\item \texttt{all} Set all trace flags except \{tracer, Tracer\} and \texttt{cpu\_timestamp} that are in their nature different than the others
\item \texttt{send} Trace sending of messages.
\item \texttt{'receive'} Trace receiving of messages.
\item \texttt{procs} Trace process related events.\end{itemize}

\begin{verbatim}
Message tags: send, send\_to\_non\_existing\_process.
\end{verbatim}

\begin{verbatim}
Message tags: 'receive'.
\end{verbatim}

\begin{verbatim}
Message tags: spawn, exit, register, unregister, link, unlink, getting\_linked, getting\_unlinked.
\end{verbatim}
call Trace certain function calls. Specify which function calls to trace by calling erlang:trace_pattern/3 [page 133].

Message tags: call, return_from.
silent Used in conjunction with the call trace flag. The call, return_from and return_to trace messages are inhibited if this flag is set, but if there are match specs they are executed as normal.
Silent mode is inhibited by executing erlang:trace(_, false, [silent|_]), or by a match spec executing the {silent, false} function.
The silent trace flag facilitates setting up a trace on many or even all processes in the system. Then the interesting trace can be activated and deactivated using the {silent, Bool} match spec function, giving a high degree of control of which functions with which arguments that triggers the trace.

Message tags: call, return_from, return_to. Or rather, the absence of.
return_to Used in conjunction with the call trace flag. Trace the actual return from a traced function back to its caller. Only works for functions traced with the local option to erlang:trace_pattern/3 [page 133].
The semantics is that a trace message is sent when a call traced function actually returns, that is, when a chain of tail recursive calls is ended. There will be only one trace message sent per chain of tail recursive calls, why the properties of tail recursiveness for function calls are kept while tracing with this flag. Using call and return_to trace together makes it possible to know exactly in which function a process executes at any time.
To get trace messages containing return values from functions, use the {return_trace} match_spec action instead.

Message tags: return_to.
running Trace scheduling of processes.

Message tags: in, out.
garbage_collection Trace garbage collections of processes.

Message tags: gc_start, gc_end.
timestamp Include a time stamp in all trace messages. The time stamp (Ts) is of the same form as returned by erlang:now().
cpu_timestamp A global trace flag for the Erlang node that makes all trace timestamps be in CPU time, not wallclock. It is only allowed with PidSpec==all. If the host machine operating system does not support high resolution CPU time measurements, trace/3 exits with badarg.
arity Used in conjunction with the call trace flag. {M, F, Arity} will be specified instead of {M, F, Args} in call trace messages.
set_on_spawn Makes any process created by a traced process inherit its trace flags, including the set_on_spawn flag.
set_on_first_spawn Makes the first process created by a traced process inherit its trace flags, excluding the set_on_first_spawn flag.
set_on_link Makes any process linked to a traced process inherit its trace flags, including the set_on_link flag.
set_on_first_link Makes the first process linked by a traced process inherit its trace flags, excluding the set_on_first_link flag.
{tracer, Tracer} Specify where to send the trace messages. Tracer must be the pid of a local process or the port identifier of a local port. If this flag is not given, trace messages will be sent to the process that called erlang:trace/3.
The effect of combining set_on_first_link with set_on_link is the same as having set_on_first_link alone. Likewise for set_on_spawn and set_on_first_spawn.

If the timestamp flag is not given, the tracing process will receive the trace messages described below. Pid is the pid of the traced process in which the traced event has occurred. The third element of the tuple is the message tag.

If the timestamp flag is given, the first element of the tuple will be trace_ts instead and the timestamp is added last in the tuple.

\{trace, Pid, 'receive', Msg\} When Pid receives the message Msg.
\{trace, Pid, send, Msg, To\} When Pid sends the message Msg to the process To.
\{trace, Pid, send_to_non_existing_process, Msg, To\} When Pid sends the message Msg to the non-existing process To.
\{trace, Pid, call, \{M, F, Args\}\} When Pid calls a traced function. The return values of calls are never supplied, only the call and its arguments. Note that the trace flag arity can be used to change the contents of this message, so that Arity is specified instead of Args.
\{trace, Pid, return_to, \{M, F, Args\}\} When Pid returns to the specified function. This trace message is sent if both the call and the return_to flags are set, and the function is set to be traced on local function calls. The message is only sent when returning from a chain of tail recursive function calls where at least one call generated a call trace message (that is, the functions match specification matched and \{message, false\} was not an action).
\{trace, Pid, return_from, \{M, F, Args\}, ReturnValue\} When Pid returns from the specified function. This trace message is sent if the call flag is set, and the function has a match specification with a return_trace or exception_trace action.
\{trace, Pid, exception_from, \{M, F, Args\}, \{Class, Value\}\} When Pid exits from the specified function due to an exception. This trace message is sent if the call flag is set, and the function has a match specification with an exception_trace action.
\{trace, Pid, spawn, Pid2, \{M, F, Args\}\} When Pid spawns a new process Pid2 with the specified function call as entry point. Note that Args is supposed to be the argument list, but may be any term in the case of an erroneous spawn.
\{trace, Pid, exit, Reason\} When Pid exits with reason Reason.
\{trace, Pid, link, Pid2\} When Pid links to a process Pid2.
\{trace, Pid, unlink, Pid2\} When Pid removes the link from a process Pid2.
\{trace, Pid, getting_linked, Pid2\} When Pid gets linked to a process Pid2.
\{trace, Pid, getting_unlinked, Pid2\} When Pid gets unlinked from a process Pid2.
\{trace, Pid, register, RegName\} When Pid gets the name RegName registered.
\{trace, Pid, unregister, RegName\} When Pid gets the name RegName unregistered. Note that this is done automatically when a registered process exits.
\{trace, Pid, in, \{M, F, Arity\}\ | 0\} When Pid is scheduled to run. The process will run in function \{M, F, Arity\}. On some rare occasions the current function cannot be determined, then the last element Arity is 0.
When Pid is scheduled out. The process was running in function \(f(M, F, \text{Arity})\). On some rare occasions the current function cannot be determined, then the last element \(\text{Arity}\) is 0.

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.

- \(\text{heap\_size}\) The size of the used part of the heap.
- \(\text{old\_heap\_size}\) The size of the used part of the old heap.
- \(\text{stack\_size}\) The actual size of the stack.
- \(\text{recent\_size}\) The size of the data that survived the previous garbage collection.
- \(\text{mbuf\_size}\) The combined size of message buffers associated with the process.

All sizes are in words.

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 \text{PidSpec}. If \text{PidSpec} is a pid, the return value will be 1. If \text{PidSpec} is all or existing the return value will be the number of processes running, excluding tracer processes. If \text{PidSpec} is new, the return value will be 0.

Failure: If specified arguments are not supported. For example \text{cpu\_timestamp} is not supported on all platforms.

\[
erlang:trace_info(PidOrFunc, Item) -> Res
\]

Types:
- PidOrFunc = pid() | new | \{Module, Function, Arity\} | onload
- Module = Function = atom()
- Arity = int()
- Item, Res - see below

Returns trace information about a process or function.

To get information about a process, \text{PidOrFunc} should be a pid or the \text{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: \text{send}, \text{'receive'}, \text{set\_on\_spawn}, \text{call}, \text{return\_to}, \text{procs}, \text{set\_on\_first\_spawn}, \text{set\_on\_link}, \text{running}, \text{garbage\_collection}, \text{timestamp}, and \text{arity}. The order is arbitrary.

- **tracer** Return the identifier for process or port tracing this process. If this process is not being traced, the return value will be \[\].
To get information about a function, `PidOrFunc` should be a three-element tuple: `{Module, Function, Arity}` or the atom `on_load`. No wildcards are allowed. Returns `undefined` if the function does not exist or `false` if the function is not traced at all. `Item` must have one of the following values:

- `traced` Return `global` if this function is traced on global function calls, `local` if this function is traced on local function calls (i.e. local and global function calls), and `false` if neither local nor global function calls are traced.

- `matchspec` Return the match specification for this function, if it has one. If the function is locally or globally traced but has no match specification defined, the returned value is `[]`.

- `meta` Return the meta trace tracer process or port for this function, if it has one. If the function is not traced but has a match specification defined, the returned value is `false`.

- `meta_matchspec` Return the meta trace match specification for this function, if it has one. If the function is meta traced but has no match specification defined, the returned value is `false`.

- `call_count` Return the call count value for this function or `true` for the pseudo function `on_load` if call count tracing is active. Return `false` otherwise. See also `erlang:trace_pattern/3` [page 133].

- `all` Return a list containing the `{Item, Value}` tuples for all other items, or return `false` if no tracing is active for this function.

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

```erlang
erlang:trace_pattern(MFA, MatchSpec) -> int()
```

The same as `erlang:trace_pattern(MFA, MatchSpec, [])` [page 133], retained for backward compatibility.

```erlang
erlang:trace_pattern(MFA, MatchSpec, FlagList) -> int()
```

Types:
- `MFA, MatchSpec, FlagList` – see below

This BIF is used to enable or disable call tracing for exported functions. It must be combined with `erlang:trace/3` [page 129] 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 129] to add or remove one or more processes to the set of traced processes. Use `erlang:trace_pattern/2` to add or remove exported functions to the set of traced functions.
The `erlang:trace_pattern/3` BIF can also add match specifications to an exported function. A match specification comprises a pattern that the arguments to the function must match, a guard expression which must evaluate to `true` and an action to be performed. The default action is to send a trace message. If the pattern does not match or the guard fails, the action will not be executed.

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

- `{Module, Function, ~_}` All exported functions of any arity named `Function` in module `Module`.
- `{Module, ~_, ~_}` All exported functions in module `Module`.
- `{~, ~_, ~_}` All exported functions in all loaded modules.

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

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

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

- `false` Disable tracing for the matching function(s). Any match specification will be removed.
- `true` Enable tracing for the matching function(s).
- `MatchSpecList` A list of match specifications. An empty list is equivalent to `true`. See the ERTS User’s Guide for a description of match specifications.
- `restart` For the `FlagList` option `call_count`: restart the existing counters. The behaviour is undefined for other `FlagList` options.
- `pause` For the `FlagList` option `call_count`: pause the existing counters. The behaviour is undefined for other `FlagList` options.

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

- `global` Turn on or off call tracing for global function calls (that is, calls specifying the module explicitly). Only exported functions will match and only global calls will generate trace messages. This is the default.
- `local` Turn on or off call tracing for all types of function calls. Trace messages will be sent whenever any of the specified functions are called, regardless of how they are called. If the `return_to` flag is set for the process, a `return_to` message will also be sent when this function returns to its caller.
- `meta | {meta, Pid}` Turn on or off meta tracing for all types of function calls. Trace messages will be sent to the tracer process or port `Pid` whenever any of the specified functions are called, regardless of how they are called. If no `Pid` is specified, `self()` is used as a default tracer process. Meta tracing traces all processes and does not care about the process trace flags set by `trace/3`, the trace flags are instead fixed to `[call, timestamp]`.

The match spec function `{return_trace}` works with meta trace and send its trace message to the same tracer process.
call_count  Starts (MatchSpec == true) or  stops (MatchSpec == false) call count tracing for all types of function calls. For every function a counter is incremented when the function is called, in any process. No process trace flags need to be activated.

If call count tracing is started while already running, the count is restarted from zero. Running counters can be paused with MatchSpec == pause. Paused and running counters can be restarted from zero with MatchSpec == restart.

The counter value can be read with erlang:trace_info/2 [page 132].

The global and local options are mutually exclusive and global is the default (if no options are specified). The call_count and meta options perform a kind of local tracing, and can also not be combined with global. A function can be either globally or locally traced. If global tracing is specified for a specified set of functions; local, meta and call count tracing for the matching set of local functions will be disabled, and vice versa.

When disabling trace, the option must match the type of trace that is set on the function, so that local tracing must be disabled with the local option and global tracing with the global option (or no option at all), and so forth.

There is no way to directly change part of a match specification list. If a function has a match specification, you can replace it with a completely new one. If you need to change an existing match specification, use the erlang:trace_info/2 [page 132] 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.

\[ \text{trunc(Number)} \rightarrow \text{int()} \]

Types:
- \( \text{Number} = \text{number()} \)

Returns an integer by the truncating Number.

\[ > \text{trunc}(6.5). \]
\[ 5 \]

Allowed in guard tests.

\[ \text{tuple_to_list(Tuple)} \rightarrow \text{[term()]} \]

Types:
- \( \text{Tuple} = \text{tuple()} \)

Returns a list which corresponds to Tuple. Tuple may contain any Erlang terms.

\[ > \text{tuple_to_list}([\text{share, \{'Ericsson B', 163}\}}]). \]
\[ [\text{share, \{'Ericsson B', 163\}}] \]

\[ \text{erlang:universaltime()} \rightarrow \text{\{Date, Time\}} \]

Types:
- \( \text{Date} = \text{\{Year, Month, Day\}} \)
- \( \text{Time} = \text{\{Hour, Minute, Second\}} \)
- \( \text{Year} = \text{Month} = \text{Day} = \text{Hour} = \text{Minute} = \text{Second} = \text{int()} \)
Returns the current date and time according to Universal Time Coordinated (UTC), also called GMT, in the form \{Year, Month, Day\}, \{Hour, Minute, Second\} if supported by the underlying operating system. If not, \texttt{erlang:universaltime()} is equivalent to \texttt{erlang:localtime()}.  

\begin{verbatim}
> erlang:universaltime().
\{\{1996,11,6\},\{14,18,43\}\}
\end{verbatim}

\texttt{erlang:universaltime_to_localtime([Date1, Time1]) -> [Date2, Time2]}

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

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

\begin{verbatim}
> erlang:universaltime_to_localtime([\{1996,11,6\},\{14,18,43\}]).
\{\{1996,11,7\},\{15,18,43\}\}
\end{verbatim}

Failure: \texttt{badarg} if \texttt{Date1} or \texttt{Time1} do not denote a valid date or time.

\texttt{unlink(Id) -> true}

Types:
- \texttt{Id = pid() | port()}

Removes the link, if there is one, between the calling process and the process or port referred to by \texttt{Id}.

Returns \texttt{true} and does not fail, even if there is no link to \texttt{Id}, or if \texttt{Id} does not exist. Once \texttt{unlink(Id)} has returned it is guaranteed that the link between the caller and the entity referred to by \texttt{Id} has no effect on the caller in the future (unless the link is setup again). If caller is trapping exits, an \{'EXIT', \texttt{Id}, \_\} message due to the link might have been placed in the callers message queue prior to the call, though. Note, the \{'EXIT', \texttt{Id}, \_\} message can be the result of the link, but can also be the result of \texttt{Id} calling \texttt{exit/2}. Therefore, it may be appropriate to cleanup the message queue when trapping exits after the call to \texttt{unlink(Id)}, as follow:

\begin{verbatim}
unlink(Id),
receive
    \{\'EXIT\', \texttt{Id}, \_\} ->
        true
after 0 ->
        true
end
\end{verbatim}
unregister(RegName) -> true

Types:
  • RegName = atom()

Removes the registered name RegName, associated with a pid or a port identifier.

> unregister(db).
true

Users are advised not to unregister system processes. Failure: badarg if RegName is not a registered name.

whereis(RegName) -> pid() | port() | undefined

Returns the pid or port identifier with the registered name RegName. Returns undefined if the name is not registered.

> whereis(db).
<0.43.0>

erlang:yield() -> true

Voluntarily let other processes (if any) get a chance to execute. Using erlang:yield() is similar to receive after 1 -> ok end, except that yield() is faster.
The error handler module defines what happens when certain types of errors occur.

Exports

```erlang
undefined_function(Module, Function, Args) -> term()
```

Types:
- `Module = Function = atom()`
- `Args = [term()]`

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

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

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

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

```erlang
undefined_lambda(Module, Fun, Args) -> term()
```

Types:
- `Module = Function = atom()`
- `Args = [term()]`

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

This function is evaluated if a call is made to `Fun(Arg1, .., ArgN)` when the module defining the fun is not loaded. The function is evaluated inside the process making the original call.

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

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

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

Changes in the code which may seem small can cause a deadlock as unforeseen consequences may occur. The use of input is dangerous in this type of code.
**error_logger**

Erlang Module

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

Initially, error_logger only has a primitive event handler, which buffers and prints the raw event messages. During system startup, the application Kernel replaces this with a standard event handler, by default one which writes nicely formatted output to tty. Kernel can also be configured so that events are logged to file instead, or not logged at all, see kernel(6) [page 30].

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

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

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

**Exports**

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

Types:
- Format = string()
- Data = [term()]

Sends a standard error event to the error logger. The Format and Data arguments are the same as the arguments of io:format/2. The event is handled by the standard event handler.
error_logger: error_logger: error_msg("An error occured in ~p-n", [a_module]).

-error_logger: error_report([a_module]).

error_report(Report) -> ok
Types:
  • Report = [{Tag, Data} | term()] | string() | term()
  • Tag = Data = term()
Sends a standard error report event to the error logger. The event is handled by the standard event handler.

warning_map() -> Tag
Types:
  • Tag = error | warning | info
Returns the current mapping for warning events. Events sent using warning_msg/1,2 or warning_report/1,2 are tagged as errors (default), warnings or info, depending on the value of the command line flag +W.
warning_msg(Format) -> ok
warning_msg(Format, Data) -> ok

Types:
  • Format = string()
  • Data = [term()]

Sends a standard warning event to the error logger. The Format and Data arguments are the same as the arguments of io:format/2. The event is handled by the standard event handler. It is tagged either as an error, warning or info, see warning_map/0 [page 141].

warning_report(Report) -> ok

Types:
  • Report = [{Tag, Data} | term()] | string() | term()
  • Tag = Data = term()

Sends a standard warning report event to the error logger. The event is handled by the standard event handler. It is tagged either as an error, warning or info, see warning_map/0 [page 141].

warning_report(Type, Report) -> ok

Types:
  • Type = term()
  • Report = [{Tag, Data} | term()] | string() | term()}
Sends a user defined warning report event to the error logger. An event handler to handle the event is supposed to have been added. The event is ignored by the standard event handler. It is tagged either as an error, warning or info, depending on the value of warning_map/0 [page 141].

\texttt{info\_msg(Format)} \rightarrow \texttt{ok}

\texttt{info\_msg(Format, Data)} \rightarrow \texttt{ok}

Types:
- \texttt{Format = string()}
- \texttt{Data = \{term()\}}

Sends a standard information event to the error logger. The Format and Data arguments are the same as the arguments of \texttt{io:format/2}. The event is handled by the standard event handler.

\begin{verbatim}
info\_report(Report) \rightarrow \texttt{ok}
\end{verbatim}

Types:
- \texttt{Report = \{\{Tag, Data\} \| \texttt{term()}\} \| \texttt{string()} \| \texttt{term()}\}
- \texttt{Tag = Data = \{term()\}}

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

\begin{verbatim}
info\_report(Type, Report) \rightarrow \texttt{ok}
\end{verbatim}

Types:
- \texttt{Type = \{term()\}}
- \texttt{Report = \{\{Tag, Data\} \| \texttt{term()}\} \| \texttt{string()} \| \texttt{term()}\}
- \texttt{Tag = Data = \{term()\}}
Sends a user defined information report event to the error logger. An event handler to handle the event is supposed to have been added. The event is ignored by the standard event handler.

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

```erlang
add_report_handler(Handler) -> Result
add_report_handler(Handler, Args) -> Result
```

Types:
- `Handler`, `Args`, `Result` - see `gen_event:add_handler/3`

Adds a new event handler to the error logger. The event handler must be implemented as a `gen_event` callback module, see `gen_event(3)`.

`Handler` is typically the name of the callback module and `Args` is an optional term (defaults to `[]`) passed to the initialization callback function `Module:init/1`. The function returns `ok` if successful.

The event handler must be able to handle the events [page 145] described below.

```erlang
delete_report_handler(Handler) -> Result
```

Types:
- `Handler`, `Result` - see `gen_event:delete_handler/3`

Deletes an event handler from the error logger by calling `gen_event:delete_handler(error_logger, Handler, [])`, see `gen_event(3)`.

```erlang
tty(Flag) -> ok
```

Types:
- `Flag = bool()`

Enables (`Flag == true`) or disables (`Flag == false`) printout of standard events to the `/dev/tty`. This is done by adding or deleting the standard event handler for output to the tty, thus calling this function overrides the value of the Kernel `error_logger` configuration parameter.

```erlang
logfile(Request) -> ok | Filename | {error, What}
```

Types:
- `Request = {open, Filename} | close | filename`
- `Filename = atom() | string()`
- `What = already_have_logfile | no_logfile | term()`

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

Enabling file logging can be used in combination with calling `tty(false)`, in order to have a silent system, where all standard events are logged to a file only. There can only be one active log file at a time. `Request` is one of:
{open, Filename} Opens the log file Filename. Returns ok if successful, or {error, already_have_logfile} if logging to file is already enabled, or an error tuple if another error occurred. For example, if Filename could not be opened.

close Closes the current log file. Returns ok, or {error, What}.

{filename} Returns the name of the log file Filename, or {error, no_logfile} if logging to file is not enabled.

Events

All event handlers added to the error logger must handle the following events. Gleader is the group leader pid of the process which sent the event, and Pid is the process which sent the event.

{error, Gleader, {Pid, Format, Data}} Generated when error_msg/1,2 or format is called.

{error_report, Gleader, {Pid, std_error, Report}} Generated when error_report/1 is called.

{error_report, Gleader, {Pid, Type, Report}} Generated when error_report/2 is called.

{warning_msg, Gleader, {Pid, Format, Data}} Generated when warning_msg/1,2 is called, provided that warnings are set to be tagged as warnings.

{warning_report, Gleader, {Pid, std_warning, Report}} Generated when warning_report/1 is called, provided that warnings are set to be tagged as warnings.

{warning_report, Gleader, {Pid, Type, Report}} Generated when warning_report/2 is called, provided that warnings are set to be tagged as warnings.

{info_msg, Gleader, {Pid, Format, Data}} Generated when info_msg/1,2 is called.

{info_report, Gleader, {Pid, std_info, Report}} Generated when info_report/1 is called.

{info_report, Gleader, {Pid, Type, Report}} Generated when info_report/2 is called.

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

SEE ALSO

gen_event(3), log_mf_h(3), kernel(6), sasl(6)
file
Erlang Module

The module file provides an interface to the file system.
On operating systems with thread support (Solaris and Windows), it is possible to let
file operations be performed in threads of their own, allowing other Erlang processes to
continue executing in parallel with the file operations. See the command line flag +A in
[erl(1)].

DATA TYPES

iodata() = iolist() | binary()
iolist() = [char() | binary() | iolist()]
io_device()
as returned by file:open/2, a process handling IO protocols

name() = string() | atom() | DeepList
DeepList = [char() | atom() | DeepList]

posix()
an atom which is named from the Posix error codes used in
Unix, and in the runtime libraries of most C compilers

time() = {{Year, Month, Day}, {Hour, Minute, Second}}
Year = Month = Day = Hour = Minute = Second = int()
Must denote a valid date and time

Exports

change_group(Filename, Gid) -> ok | {error, Reason}
Types:
- Filename = name()
- Gid = int()
- Reason = posix()
Changes group of a file. See write_file_info/2 [page 163].

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

Types:
- Filename = name()
- Uid = int()
- Gid = int()
- Reason = posix()

Changes owner of a file. See write_file_info/2 [page 163].

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

Types:
- Filename = name()
- Mtime = time()
- Reason = posix()

Changes the modification and access times of a file. See write_file_info/2 [page 163].

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

Types:
- Filename = name()
- Mtime = Atime = time()
- Reason = posix()

Changes the modification and last access times of a file. See write_file_info/2 [page 163].

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

Types:
- IoDevice = io_device()
- Reason = posix()

Closes the file referenced by IoDevice. It mostly returns ok, expect for some severe errors such as out of memory.

Note that if the option delayed_write was used when opening the file, close/1 might return an old write error and not even try to close the file. See open/2 [page 152].

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

Types:
- Filename = name()
- Terms = [term()]
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term - see below

Reads Erlang terms, separated by ".", from Filename. Returns one of the following:
{ok, Terms} The file was successfully read.
{error, Posix} An error occurred when opening the file or reading it. See open/2 [page 152] for a list of typical error codes.
{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang terms in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

Example:

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

copy(Source, Destination) ->
copy(Source, Destination, ByteCount) -> {ok, BytesCopied} | {error, Reason}

Types:
- Source = Destination = io_device() | Filename | {Filename, Modes}
- Filename = name()
- Modes = [Mode] - see open/2
- ByteCount = int() > = 0 | infinity
- BytesCopied = int()

Copies ByteCount bytes from Source to Destination. Source and Destination refer to either filenames or IO devices from e.g. open/2. ByteCount defaults infinity, denoting an infinite number of bytes.

The argument Modes is a list of possible modes, see open/2 [page 152], and defaults to [].

If both Source and Destination refer to filenames, the files are opened with [read, binary] and [write, binary] prepended to their mode lists, respectively, to optimize the copy.

If Source refers to a filename, it is opened with read mode prepended to the mode list before the copy, and closed when done.

If Destination refers to a filename, it is opened with write mode prepended to the mode list before the copy, and closed when done.

Returns {ok, BytesCopied} where BytesCopied is the number of bytes that actually was copied, which may be less than ByteCount if end of file was encountered on the source. If the operation fails, {error, Reason} is returned.

Typical error reasons: As for open/2 if a file had to be opened, and as for read/2 and write/2.

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

Types:
- Dir = name()
- Reason = posix()
Tries to delete the directory Dir. The directory must be empty before it can be deleted. Returns ok if successful.

Typical error reasons are:

eacces Missing search or write permissions for the parent directories of Dir.
eexist The directory is not empty.
enoent The directory does not exist.
enotdir A component of Dir is not a directory. On some platforms, enoent is returned instead.
eINVAL Attempt to delete the current directory. On some platforms, eacces is returned instead.

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

definition:
  • Filename = name()
  • Reason = posix()

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

Typical error reasons are:

  enoent The file does not exist.
eacces Missing permission for the file or one of its parents.
eperm The file is a directory and the user is not super-user.
enotdir A component of the file name is not a directory. On some platforms, enoent is returned instead.

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

definition:
  • Filename = name()
  • Reason = posix() | {Line, Mod, Term}
  • Line, Mod, Term - see below

Reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from Filename. The actual result of the evaluation is not returned; any expression sequence in the file must be there for its side effect. Returns one of the following:

  ok The file was read and evaluated.
{error, Posix} An error occurred when opening the file or reading it. See open/2 for a list of typical error codes.
{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang expressions in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

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

definition:
  • Filename = name()
  • Reason = posix() | {Line, Mod, Term}
  • Line, Mod, Term - see below

Kernel Application (KERNEL)
- Filename = name()
- Bindings – see erl eval(3)
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see eval/1

The same as eval/1 but the variable bindings Bindings are used in the evaluation. See [erl eval(3)] about variable bindings.

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

This function is obsolete. Use read_file_info/1 instead.

```erlang
format_error(Reason) -> Chars
```

Types:
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see eval/1
- Chars = [char()] | Chars

Given the error reason returned by any function in this module, returns a descriptive string of the error in English.

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

Types:
- Dir = string()
- Reason = posix()

Returns {ok, Dir}, where Dir is the current working directory of the file server.

**Note:**
In rare circumstances, this function can fail on Unix. It may happen if read permission does not exist for the parent directories of the current directory.

Typical error reasons are:
- eaccess Missing read permission for one of the parents of the current directory.

```erlang
get_cwd(Drive) -> {ok, Dir} | {error, Reason}
```

Types:
- Drive = string() – see below
- Dir = string()
- Reason = posix()

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

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

Typical error reasons are:
The operating system have no concept of drives.

The drive does not exist.

The format of Drive is invalid.

\[\text{list}\{\text{Dir}(\text{Dir}) \rightarrow \{\text{ok, Filenames}\} | \{\text{error, Reason}\}\]

Types:
- \text{Dir} = \text{name()}
- \text{Filenames} = \{\text{Filename}\}
- \text{Filename} = \text{string()}
- \text{Reason} = \text{posix()}

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

Typical error reasons are:
- \text{eacces} Missing search or write permissions for \text{Dir} or one of its parent directories.
- \text{enoent} The directory does not exist.

\[\text{make}\{\text{dir}(\text{Dir}) \rightarrow \text{ok} | \{\text{error, Reason}\}\]

Types:
- \text{Dir} = \text{name()}
- \text{Reason} = \text{posix()}

Tries to create the directory \text{Dir}. Missing parent directories are not created. Returns \text{ok} if successful.

Typical error reasons are:
- \text{eacces} Missing search or write permissions for the parent directories of \text{Dir}.
- \text{eexist} \text{New} already exists.
- \text{enoent} A component of \text{Dir} does not exist.
- \text{enospc} There is a no space left on the device.
- \text{enotdir} A component of \text{Dir} is not a directory. On some platforms, \text{enoent} is returned instead.

\[\text{make}\{\text{link}(\text{Existing, New}) \rightarrow \text{ok} | \{\text{error, Reason}\}\]

Types:
- \text{Existing = New} = \text{name()}
- \text{Reason} = \text{posix()}

Makes a hard link from \text{Existing} to \text{New}, on platforms that support links (Unix). This function returns \text{ok} if the link was successfully created, or \{\text{error, Reason}\}. On platforms that do not support links, \{\text{error, enotsup}\} is returned.

Typical error reasons:
- \text{eacces} Missing read or write permissions for the parent directories of \text{Existing or New}.
- \text{eexist} \text{New} already exists.
enotsup Hard links are not supported on this platform.

make_symlink(Name1, Name2) -> ok | {error, Reason}

Types:
- Name1 = Name2 = name()
- Reason = posix()

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

Typical error reasons:
- eacces Missing read or write permissions for the parent directories of Name1 or Name2.
- exexist Name2 already exists.
- enotsup Symbolic links are not supported on this platform.

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

Types:
- Filename = name()
- Modes = [Mode]
- Mode = read | write | append | binary | {delayed_write, Size, Delay} | delayed_write | {read_ahead, Size} | read_ahead | compressed
- Size = Delay = int()
- IoDevice = io_device()
- Reason = posix()

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

read The file, which must exist, is opened for reading.
write The file is opened for writing. It is created if it does not exist. If the file exists, and if write is not combined with read, the file will be truncated.
append The file will be opened for writing, and it will be created if it does not exist. Every write operation to a file opened with append will take place at the end of the file.
raw The raw option allows faster access to a file, because no Erlang process is needed to handle the file. However, a file opened in this way has the following limitations:
  - The functions in the io module cannot be used, because they can only talk to an Erlang process. Instead, use the read/2 and write/2 functions.
  - Only the Erlang process which opened the file can use it.
  - A remote Erlang file server cannot be used; the computer on which the Erlang node is running must have access to the file system (directly or through NFS).
binary This option can only be used if the raw option is specified as well. When specified, read operations on the file using the read/2 function will return binaries rather than lists.
{delayed_write, Size, Delay} If this option is used, the data in subsequent write/2 calls is buffered until there are at least Size bytes buffered, or until the oldest buffered data is Delay milliseconds old. Then all buffered data is written in one operating system call. The buffered data is also flushed before some other file operation than write/2 is executed.

The purpose of this option is to increase performance by reducing the number of operating system calls, so the write/2 calls should be for sizes significantly less than Size, and not interspersed by 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.

For example, when delayed_write is used, after a number of write/2 calls close/1 might return {error, enospc} because there was not enough space on the disc for previously written data, and close/1 should probably be called again since the file is still open.

delayed_write The same as {delayed_write, Size, Delay} with reasonable default values for Size and Delay. (Roughly some 64 KBytes, 2 seconds)

{read_ahead, Size} This option activates read data buffering. If read/2 calls are for significantly less than Size bytes, read operations towards the operating system are still performed for blocks of Size bytes. The extra data is buffered and returned in subsequent read/2 calls, giving a performance gain since the number of operating system calls is reduced.

If read/2 calls are for sizes not significantly less than, or even greater than Size bytes, no performance gain can be expected.

read_ahead The same as {read_ahead, Size} with a reasonable default value for Size. (Roughly some 64 KBytes)

compressed Makes it possible to read and write gzip compressed files. Note that the file size obtained with read_file_info/1 will most probably not match the number of bytes that can be read from a compressed file.

Returns:

{ok, IoDevice} The file has been opened in the requested mode. IoDevice is a reference to the file.

{error, Reason} The file could not be opened.

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

Note:

In previous versions of file, modes were given as one of the atoms read, write, or read_write instead of a list. This is still allowed for reasons of backwards compatibility, but should not be used for new code. Also note that read_write is not allowed in a mode list.

Typical error reasons:
enoent  The file does not exist.
eaccess Missing permission for reading the file or searching one of the parent directories.
eisdir  The named file is not a regular file. It may be a directory, a fifo, or a device.
enotdir A component of the file name is not a directory. On some platforms, enoent is returned instead.
enospc  There is no space left on the device (if write access was specified).

path_consult(Path, Filename) -> {ok, Terms, FullName} | {error, Reason}

Types:
- Path = [Dir]
- Dir = name()
- Filename = name()
- Terms = [term()]
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute filename, Path is ignored. Then reads Erlang terms, separated by '.', from the file. Returns one of the following:

{ok, Terms, FullName} The file was successfully read. FullName is the full name of the file.
{error, enoent} The file could not be found in any of the directories in Path.
{error, Posix} An error occurred when opening the file or reading it. See open/2 [page 152] for a list of typical error codes.
{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang terms in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

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

Types:
- Path = [Dir]
- Dir = name()
- Filename = name()
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute file name, Path is ignored. Then reads and evaluates Erlang expressions, separated by '.', (or ',', a sequence of expressions is also an expression), from the file. The actual result of evaluation is not returned; any expression sequence in the file must be there for its side effect. Returns one of the following:

{ok, FullName} The file was read and evaluated. FullName is the full name of the file.
path_open(Path, Filename, Modes) -> {ok, IoDevice, FullName} | {error, Reason}

Types:
- Path = [Dir]
- Dir = name()
- Filename = name()
- Modes = [Mode] – see open/2
- IoDevice = io_device()
- FullName = string()
- Reason = posix()

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute file name, Path is ignored. Then opens the file in the mode determined by Modes. Returns one of the following:

- {ok, IoDevice, FullName} The file has been opened in the requested mode. IoDevice is a reference to the file and FullName is the full name of the file.
- {error, enoent} The file could not be found in any of the directories in Path.
- {error, Posix} An error occurred when opening the file or reading it. See open/2 [page 152] for a list of typical error codes.

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

Types:
- Path = [Dir]
- Dir = name()
- Filename = name()
- Value = term()
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute file name, Path is ignored. Then reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. Returns one of the following:

- {ok, Value, FullName} The file was read and evaluated. FullName is the full name of the file and Value the value of the last expression.
- {error, enoent} The file could not be found in any of the directories in Path.
- {error, Posix} An error occurred when opening the file or reading it. See open/2 [page 152] for a list of typical error codes.
{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang expressions in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

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

Types:
- Path = [Dir]
- Dir = name()
- Filename = name()
- Bindings - see erl_eval(3)
- Value = term()
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term - see path_script/2

The same as path_script/2 but the variable bindings Bindings are used in the evaluation. See [erl_eval(3)] about variable bindings.

pid2name(Pid) -> string() | undefined

Types:
- Pid = pid()

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

{ok, Filename} If this node’s file server is not a slave, the file was opened by this node’s file server, (this implies that Pid must be a local pid) and the file is not closed. Filename is the filename in flat string format.

undefined In all other cases.

**Warning:**
This function is intended for debugging only.

position(IoDevice, Location) -> {ok, NewPosition} | {error, Reason}

Types:
- IoDevice = io_device()
- Location = Offset | {bof, Offset} | {cur, Offset} | {eof, Offset} | bof | cur | eof
- Offset = int()
- NewPosition = int()
- Reason = posix()

Sets the position of the file referenced by IoDevice to Location. Returns {ok, NewPosition} (as absolute offset) if successful, otherwise {error, Reason}. Location is one of the following:

Offset The same as {bof, Offset}. 
Kernel Reference Manual

{bof, Offset} Absolute offset.
{cur, Offset} Offset from the current position.
{eof, Offset} Offset from the end of file.
bof | cur | eof The same as above with Offset 0.

Typical error reasons are:
einval Either Location was illegal, or it evaluated to a negative offset in the file. Note that if the resulting position is a negative value, the result is an error, and after the call the file position is undefined.

pread(IoDevice, LocNums) -> {ok, DataL} | {error, Reason}

Types:
• IoDevice = io_device()
• LocNums = [{Location, Number}]
• Location – see position/2
• Number = int()
• DataL = [Data]
• Data = [char()] | binary() | eof
• Reason = posix() 

Performs a sequence of pread/3 in one operation, which is more efficient than calling them one at a time. Returns {ok, [Data, ...]} or {error, Reason}, where each Data, the result of the corresponding pread, is either a list or a binary depending on the mode of the file, or eof if the requested position was beyond end of file.

pread(IoDevice, Location, Number) -> {ok, Data} | {error, Reason}

Types:
• IoDevice = io_device()
• Location – see position/2
• Number = int()
• Data = [char()] | binary() | eof
• Reason = posix() 

Combines position/2 and read/2 in one operation, which is more efficient than calling them one at a time. If IoDevice has been opened in raw mode, some restrictions apply: Location is only allowed to be an integer; and the current position of the file is undefined after the operation.

pwrite(IoDevice, LocBytes) -> ok | {error, {N, Reason}}

Types:
• IoDevice = io_device()
• LocBytes = [{Location, Bytes}]
• Location – see position/2
• Bytes = iodata()
• N = int()
• Reason = posix()
Performs a sequence of `pwrite/3` in one operation, which is more efficient than calling them one at a time. Returns `ok` or 
`{error, {N, Reason}}`, where N is the number of successful writes that was done before the failure.

```plaintext
pwrite(IoDevice, Location, Bytes) -> ok | {error, Reason}
Types:
  • IoDevice = io_device()
  • Location – see position/2
  • Bytes = iodata()
  • Reason = posix()
Combines `position/2` and `write/2` in one operation, which is more efficient than calling them one at a time. If `IoDevice` has been opened in raw mode, some restrictions apply: `Location` is only allowed to be an integer; and the current position of the file is undefined after the operation.
```

```plaintext
read(IoDevice, Number) -> ok, Data | eof | {error, Reason}
Types:
  • IoDevice = io_device()
  • Number = int()
  • Data = [char()] | binary()
  • Reason = posix()
Reads Number bytes from the file referenced by `IoDevice`. This function is the only way to read from a file opened in raw mode (although it works for normally opened files, too). Returns:

  {ok, Data} If the file was opened in binary mode, the read bytes are returned in a binary, otherwise in a list. The list or binary will be shorter than the number of bytes requested if end of file was reached.
  eof Returned if `Number>0` and end of file was reached before anything at all could be read.
  {error, Reason} An error occurred.
Typical error reasons:
  ebadf The file is not opened for reading.
```

```plaintext
read_file(Filename) -> {ok, Binary} | {error, Reason}
Types:
  • Filename = name()
  • Binary = binary()
  • Reason = posix()
Returns `{ok, Binary}`, where `Binary` is a binary data object that contains the contents of `Filename`, or `{error, Reason}` if an error occurs.
Typical error reasons:
  enoent The file does not exist.
```
eacces  Missing permission for reading the file, or for searching one of the parent
directories.
eisdir  The named file is a directory.
enotdir A component of the file name is not a directory. On some platforms, enoent
is returned instead.
enomem There is not enough memory for the contents of the file.

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

Types:
  • Filename = name()
  • FileInfo = #file_info{}
  • Reason = posix()

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise
{error, Reason}. FileInfo is a record file_info, defined in the Kernel include file
file.hrl. Include the following directive in the module from which the function is
called:

-include_lib("kernel/include/file.hrl").

The record file_info contains the following fields.

  size = int() Size of file in bytes.
  type = device | directory | regular | other The type of the file.
  access = read | write | read_write | none The current system access to the file.
  atime = time() The last (local) time the file was read.
  mtime = time() The last (local) time the file was written.
  ctime = time() The interpretation of this time field depends on the operating system.
          On Unix, it is the last time the file or the inode was changed. In Windows, it is the
          create time.
  mode = int() The file permissions as the sum of the following bit values:
         8#00400 read permission: owner
         8#00200 write permission: owner
         8#00100 execute permission: owner
         8#00040 read permission: group
         8#00020 write permission: group
         8#00010 execute permission: group
         8#00004 read permission: other
         8#00002 write permission: other
         8#00001 execute permission: other
         16#800 set user id on execution
         16#400 set group id on execution

          On Unix platforms, other bits than those listed above may be set.
  links = int() Number of links to the file (this will always be 1 for file systems which
          have no concept of links).
  major_device = int() Identifies the file system where the file is located. In Windows,
          the number indicates a drive as follows: 0 means A:, 1 means B:, and so on.
minor_device = int() Only valid for character devices on Unix. In all other cases, this field is zero.

inode = int() Gives the inode number. On non-Unix file systems, this field will be zero.

uid = int() Indicates the owner of the file. Will be zero for non-Unix file systems.

gid = int() Gives the group that the owner of the file belongs to. Will be zero for non-Unix file systems.

Typical error reasons:

- eacces Missing search permission for one of the parent directories of the file.
- enoent The file does not exist.
- enotdir A component of the file name is not a directory. On some platforms, enoent is returned instead.

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

Types:
- Name = name()
- Filename = string()
- Reason = posix()

This function returns {ok, Filename} if Name refers to a symbolic link or {error, Reason} otherwise. On platforms that do not support symbolic links, the return value will be {error, enotsup}.

Typical error reasons:

- einval Linkname does not refer to a symbolic link.
- enoent The file does not exist.
- enotsup Symbolic links are not supported on this platform.

read_link_info(Name) -> {ok, FileInfo} | {error, Reason}

Types:
- Name = name()
- FileInfo = #file_info, see read_file_info/1
- Reason = posix()

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

If Name is not a symbolic link, this function returns exactly the same result as read_file_info/1. On platforms that do not support symbolic links, this function is always equivalent to read_file_info/1.

rename(Source, Destination) -> ok | {error, Reason}

Types:
- Source = Destination = name()
- Reason = posix()
Tries to rename the file Source to Destination. It can be used to move files (and directories) between directories, but it is not sufficient to specify the destination only. The destination file name must also be specified. For example, if bar is a normal file and foo and baz are directories, rename("foo/bar", "baz") returns an error, but rename("foo/bar", "baz/bar") succeeds. Returns ok if it is successful.

**Note:**
Renaming of open files is not allowed on most platforms (see *eacces* below).

Typical error reasons:

eacces  Missing read or write permissions for the parent directories of Source or Destination. On some platforms, this error is given if either Source or Destination is open.
eexist  Destination is not an empty directory. On some platforms, also given when Source and Destination are not of the same type.
eINVAL  Source is a root directory, or Destination is a sub-directory of Source.
eISDIR  Destination is a directory, but Source is not.
eNODIR  Source does not exist.
eNODIR  Source is a directory, but Destination is not.
exdev  Source and Destination are on different file systems.

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

Types:
  • Filename = name()
  • Value = term()
  • Reason = posix() | {Line, Mod, Term}
  • Line, Mod, Term – see below

Reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. Returns one of the following:

{ok, Value} The file was read and evaluated. Value is the value of the last expression.
{error, Posix} An error occurred when opening the file or reading it. See open/2 [page 152] for a list of typical error codes.
{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang expressions in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.
```

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

Types:
  • Filename = name()
  • Bindings – see erl_eval(3)
  • Value = term()
  • Reason = posix() | {Line, Mod, Term}
```
The same as script/1 but the variable bindings Bindings are used in the evaluation. See [erl_eval(3)] about variable bindings.

**set_cwd(Dir) -> ok**

Types:
- Dir = name()

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

Typical error reasons are:
- enoent The directory does not exist.
- enotdir A component of Dir is not a directory. On some platforms, enoent is returned.
- eaccess Missing permission for the directory or one of its parents.

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

Types:
- IoDevice = io_device()
- Reason = posix()

Makes sure that any buffers kept by the operating system (not by the Erlang runtime system) are written to disk. On some platforms, this function might have no effect.

Typical error reasons are:
- enospc Not enough space left to write the file.

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

Types:
- IoDevice = io_device()
- Reason = posix()

Truncates the file referenced by IoDevice at the current position. Returns ok if successful, otherwise {error, Reason}.

**write(IoDevice, Bytes) -> ok | {error, Reason}**

Types:
- IoDevice = io_device()
- Bytes = iodata()
- Reason = posix()

Writes Bytes to the file referenced by IoDevice. This function is the only way to write to a file opened in raw mode (although it works for normally opened files, too). Returns ok if successful, and {error, Reason} otherwise.

Typical error reasons are:
- ebadf The file is not opened for writing.
- enospc There is no space left on the device.
write_file(Filename, Binary) -> ok | {error, Reason}

Types:
- Filename = name()
- Binary = binary()
- Reason = posix()

Writes the contents of the binary data object Binary to the file Filename. The file is created if it does not exist. If it exists, the previous contents are overwritten. Returns ok, or {error, Reason}.

Typical error reasons are:
- enoent A component of the file name does not exist.
- enotdir A component of the file name is not a directory. On some platforms, enoent is returned instead.
- enospc There is 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(Filename, Binary, Modes) -> ok | {error, Reason}

Types:
- Filename = name()
- Binary = binary()
- Modes = [Mode] – see open/2
- Reason = posix()

Same as write_file/2, but takes a third argument Modes, a list of possible modes, see open/2 [page 152]. The mode flags binary and write are implicit, so they should not be used.

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

Types:
- Filename = name()
- FileInfo = #file_info() – see also read_file_info/1
- Reason = posix()

Change file information. Returns ok if successful, otherwise {error, Reason}. FileInfo is a record file_info, defined in the Kernel include file file.hrl. Include the following directive in the module from which the function is called:
-include_lib("kernel/include/file.hrl").

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

- atime = time() The last (local) time the file was read.
- mtime = time() The last (local) time the file was written.
- ctime = time() On Unix, any value given for this field will be ignored (the “ctime” for the file will be set to the current time). On Windows, this field is the new creation time to set for the file.
- mode = int() The file permissions as the sum of the following bit values:
read permission: owner
write permission: owner
execute permission: owner
read permission: group
write permission: group
execute permission: group
read permission: other
write permission: other
execute permission: other
set user id on execution
set group id on execution

On Unix platforms, other bits than those listed above may be set.

uid = int() Indicates the owner of the file. Ignored for non-Unix file systems.

gid = int() Gives the group that the owner of the file belongs to. Ignored non-Unix file systems.

Typical error reasons:

eacces  Missing search permission for one of the parent directories of the file.
enoent  The file does not exist.
enotdir A component of the file name is not a directory. On some platforms, enoent is returned instead.

POSIX Error Codes

- eacces  - permission denied
- eagain  - resource temporarily unavailable
- ebadf   - bad file number
- ebusy   - file busy
- edquot  - disk quota exceeded
- eexist  - file already exists
-efault  - bad address in system call argument
- efbig   - file too large
- eintr   - interrupted system call
- einval  - invalid argument
- eloop   - too many levels of symbolic links
- emfile  - too many open files
- emlink  - too many links
- enametoolong - file name too long
- enfie  - file table overflow
- enodev  - no such device
- enoent - no such file or directory
- enomem - not enough memory
- enospc - no space left on device
- enotblk - block device required
- enotdir - not a directory
- enotsup - operation not supported
- enxio - no such device or address
- eperm - not owner
- epipe - broken pipe
- erofs - read-only file system
- espipe - invalid seek
- esrch - no such process
- estale - stale remote file handle
- exdev - cross-domain link

Performance

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

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

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

For increased performance, raw files are recommended. Raw files uses the file system of the node's host machine. For normal files (non-raw), the file server is used to find the files, and if the node is running its file server as slave to another node's, and the other node runs on some other host machine, they may have different file systems. This is seldom a problem, but you have now been warned.

A normal file is really a process so it can be used as an IO device (see `io`). Therefore when data is written to a normal file, the sending of the data to the file process, copies all data that are not binaries. Opening the file in binary mode and writing binaries is therefore recommended. If the file is opened on another node, or if the file server runs as slave to another node's, also binaries are copied.

Caching data to reduce the number of file operations, or rather the number of calls to the file driver, will generally increase performance. The following function writes 4 MBytes in 23 seconds when tested:
create_file_slow(Name, N) when integer(N), N >= 0 ->
  {ok, FD} = file:open(Name, [raw, write, delayed_write, binary]),
  ok = create_file_slow(FD, 0, N),
  ok = ?FILE_MODULE:close(FD),
  ok.
create_file_slow(FD, M, M) ->
  ok;
create_file_slow(FD, M, N) ->
  ok = file:write(FD, &lt;&lt;M:32/unsigned&gt;&gt;),
  create_file_slow(FD, M+1, N).

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

create_file(Name, N) when integer(N), N >= 0 ->
  {ok, FD} = file:open(Name, [raw, write, delayed_write, binary]),
  ok = create_file(FD, 0, N),
  ok = ?FILE_MODULE:close(FD),
  ok.
create_file(FD, M, M) ->
  ok;
create_file(FD, M, N) when M + 1024 =&lt; N ->
  create_file(FD, M, M + 1024, []),
  create_file(FD, M + 1024, N);
create_file(FD, M, N) ->
  create_file(FD, M, N, []).
Note:
Trust only your own benchmarks. If the list length in create_file/2 above is increased, it will run slightly faster, but consume more memory and cause more memory fragmentation. How much this affects your application is something that this simple benchmark can not predict.

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

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

A raw file is really a port. When writing data to a port, it is efficient to write a list of binaries. There is no need to flatten a deep list before writing. On Unix hosts, scatter output, which writes a set of buffers in one operation, is used when possible. In this way file:write(FD, [Bin1, Bin2 | Bin3]) will write the contents of the binaries without copying the data at all except for perhaps deep down in the operating system kernel.

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

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

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

Warnings

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

SEE ALSO

[filename(3)]
The `gen_tcp` module provides functions for communicating with sockets using the TCP/IP protocol.

The following code fragment provides a simple example of a client connecting to a server at port 5678, transferring a binary and closing the connection:

```erlang
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:

```erlang
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.
```

```erlang
do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
            do_recv(Sock, [Bs, B]);
        {error, closed} ->
            {ok, list_to_binary(Bs)}
    end.
```

**DATA TYPES**

- `ip_address()`
  
    see inet(3)

- `posix()`
  
    see inet(3)

- `socket()`
  
    as returned by accept/1,2 and connect/3,4
Exports

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

Types:
- Address = string() | atom() | ip_address()
- Port = 0..65535
- Options = [Opt]
  - Opt - see below
  - Timeout = int() | infinity
- Socket = socket()
- Reason = posix()

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

- list Received Packet is delivered as a list.
- binary Received Packet is delivered as a binary.
  - {ip, ip_address()} If the host has several network interfaces, this option specifies which one to use.
  - {port, Port} Specify which local port number to use.
  - {fd, int()} If a socket has somehow been connected without using gen_tcp, use this option to pass the file descriptor for it.
- inet6 Set up the socket for IPv6.
- inet Set up the socket for IPv4.
- Opt See inet:setopts/2 [page 190].

Packets can be sent to the returned socket Socket using send/2. Packets sent from the peer are delivered as messages:

- {tcp, Socket, Data}

If the socket is closed, the following message is delivered:

- {tcp_closed, Socket}

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

- {tcp_error, Socket, Reason}

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

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

Note:
The default values for options given to connect can be affected by the Kernel configuration parameter inet_default_connect_options. See inet(3) [page 187] for details.
listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}

Types:
- Port = 0..65535
- Options = [Opt]
- Opt - see below
- ListenSocket - see below
- Reason = posix()

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

If Port == 0, the underlying OS assigns an available port number, use inet:port/1 to retrieve it.

The available options are:

- list Received Packet is delivered as a list.
- binary Received Packet is delivered as a binary.
- {backlog, B} B is an integer >= 0. The backlog value defaults to 5. The backlog value defines the maximum length that the queue of pending connections may grow to.
- {ip, ip_address()} If the host has several network interfaces, this option specifies which one to listen on.
- {fd, Fd} If a socket has somehow been connected without using gen_tcp, use this option to pass the file descriptor for it.
- inet6 Set up the socket for IPv6.
- inet Set up the socket for IPv4.
- Opt See inet:setopts/2 [page 190].

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

Note:
The default values for options given to listen can be affected by the Kernel configuration parameter inet.default.listen.options. See inet(3) [page 187] for details.

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

Types:
- ListenSocket - see listen/2
- Timeout = int() | infinity
- Socket = socket()
- Reason = closed | timeout | posix()
Accepts an incoming connection request on a listen socket. Socket must be a socket returned from \texttt{listen/2}. Timeout specifies a timeout value in ms, defaults to infinity.

Returns \{ok, \textit{Socket}\} if a connection is established, or \{error, \textit{closed}\} if \textit{ListenSocket} is closed, or \{error, \textit{timeout}\} if no connection is established within the specified time. May also return a POSIX error value if something else goes wrong, see \texttt{inet(3)} for possible error values.

Packets can be sent to the returned socket \textit{Socket} using \texttt{send/2}. Packets sent from the peer are delivered as messages:

\{tcp, \textit{Socket}, \textit{Data}\}

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

\begin{verbatim}
\textbf{send(Socket, Packet)} \rightarrow \textit{ok} | \{error, \textit{Reason}\}
\end{verbatim}

\textbf{Types:}

- \texttt{Socket = socket()}
- \texttt{Packet = [char()] | binary()}
- \texttt{Reason = posix()}

Sends a packet on a socket.

\begin{verbatim}
\textbf{recv(Socket, Length)} \rightarrow \{ok, \textit{Packet}\} | \{error, \textit{Reason}\}
\textbf{recv(Socket, Length, Timeout)} \rightarrow \{ok, \textit{Packet}\} | \{error, \textit{Reason}\}
\end{verbatim}

\textbf{Types:}

- \texttt{Socket = socket()}
- \texttt{Length = int()}
- \texttt{Packet = [char()] | binary()}
- \texttt{Timeout = int() | infinity}
- \texttt{Reason = closed | posix()}

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

The \texttt{Length} argument is only meaningful when the socket is in raw mode and denotes the number of bytes to read. If \texttt{Length = 0}, all available bytes are returned. If \texttt{Length > 0}, exactly \texttt{Length} bytes are returned, or an error; possibly discarding less than \texttt{Length} bytes of data when the socket gets closed from the other side.

The optional \texttt{Timeout} parameter specifies a timeout in milliseconds. The default value is \texttt{infinity}.

\begin{verbatim}
\textbf{controlling process(Socket, Pid)} \rightarrow \textit{ok} | \{error, \textit{eperm}\}
\end{verbatim}

\textbf{Types:}

- \texttt{Socket = socket()}
- \texttt{Pid = pid()}

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

\begin{verbatim}
\textbf{close(Socket)} \rightarrow \textit{ok} | \{error, \textit{Reason}\}
\end{verbatim}
Closes a TCP socket.

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

**Types:**
- `Socket = socket()`
- `Reason = posix()`

Immediately close a socket in one or two directions.

- `How == write` means closing the socket for writing, reading from it is still possible.
- To be able to handle that the peer has done a shutdown on the write side, the `{exit_on_close, false}` option is useful.
The gen_udp module provides functions for communicating with sockets using the UDP protocol.

**DATA TYPES**

- **ip_address()**
  
  see inet(3)

- **posix()**
  
  see inet(3)

- **socket()**
  
  as returned by open/1,2

**Exports**

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

Types:
- Port = 0..65535
- Options = [Opt]
- Opt - see below
- Socket = socket()
- Reason = posix()

Associates a UDP port number (Port) with the calling process. The available options are:

- **list** Received Packet is delivered as a list.
- **binary** Received Packet is delivered as a binary.
- **{ip, ip_address()}** If the host has several network interfaces, this option specifies which one to use.
- **{fd, int()}** If a socket has somehow been opened without using gen_udp, use this option to pass the file descriptor for it.
- **inet6** Set up the socket for IPv6.
- **inet** Set up the socket for IPv4.
The returned socket \texttt{Socket} is used to send packets from this port with \texttt{send/4}. When UDP packets arrive at the opened port, they are delivered as messages:

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

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

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

Default value for the receive buffer option is \{\texttt{recbuf}, 8192\}.

If Port == 0, the underlying OS assigns a free UDP port, use \texttt{inet:port/1} to retrieve it.

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

Types:
\begin{itemize}
\item Socket = socket()
\item Address = string() | atom() | ip_address()
\item Port = 0..65535
\item Packet = [char()] | binary()
\item Reason = not_owner | posix()
\end{itemize}

Sends a packet to the specified address and port. The \texttt{Address} argument can be either a hostname, or an IP address.
\end{verbatim}

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

Types:
\begin{itemize}
\item Socket = socket()
\item Length = int()
\item Address = ip_address()
\item Port = 0..65535
\item Packet = [char()] | binary()
\item Timeout = int() | infinity
\item Reason = not_owner | posix()
\end{itemize}

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

The optional \texttt{Timeout} parameter specifies a timeout in milliseconds. The default value is \texttt{infinity}.
\end{verbatim}

\begin{verbatim}
controlling_process(Socket, Pid) -> ok

Types:
\begin{itemize}
\item Socket = socket()
\item Pid = pid()
\end{itemize}

Assigns a new controlling process \texttt{Pid} to \texttt{Socket}. The controlling process is the process which receives messages from the socket.
\end{verbatim}
close(Socket) -> ok | {error, Reason}

Types:
- Socket = socket()
- Reason = not_owner | posix()

Closes a UDP socket.
This documentation describes the Global module which consists of the following functionalities:

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

These services are controlled via the process `global_name_server` which exists on every node. The global name server is started automatically when a node is started. With the term `global` is meant over a system consisting of several Erlang nodes.

The ability to globally register names is a central concept in the programming of distributed Erlang systems. In this module, the equivalent of the `register/2` and `whereis/1` BIFs (for local name registration) are implemented, but for a network of Erlang nodes. A registered name is an alias for a process identifier (pid). The global name server monitors globally registered pids. If a process terminates, the name will also be globally unregistered.

The registered names are stored in replica global name tables on every node. There is no central storage point. Thus, the translation of a name to a pid is fast, as it is always done locally. When any action in taken which results in a change to the global name table, all tables on other nodes are automatically updated.

Global locks have lock identities and are set on a specific resource. For instance, the specified resource could be a pid. When a global lock is set, access to the locked resource is denied for all other resources other than the lock requester.

Both the registration and lock functionalities are atomic. All nodes involved in these actions will have the same view of the information.

The global name server also performs the critical task of continuously monitoring changes in node configuration: if a node which runs a globally registered process goes down, the name will be globally unregistered. To this end the global name server subscribes to `nodeup` and `nodedown` messages sent from the `net_kernel` module. Relevant Kernel application variables in this context are `net_setuptime`, `net_ticktime`, and `dist_auto_connect`. See also kernel(6) [page 32].

The server will also maintain a fully connected network. For example, if node N1 connects to node N2 (which is already connected to N3), the global name servers on the nodes N1 and N3 will make sure that also N1 and N3 are connected. If this is not desired, the command line flag `-connect all false` can be used (see also [erl(1)]). In this case the name registration facility cannot be used, but the lock mechanism will still work.

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

**Note:**
If the fully connected network is not set up properly, the first thing to try is to increase the value of `net_setuptime`.

### Exports

**del_lock(Id)**
**del_lock(Id, Nodes) -> void()**

Types:
- `Id` = {ResourceId, LockRequesterId}
- `ResourceId` = term()
- `LockRequesterId` = term()
- `Nodes` = [node()]

Deletes the lock `Id` synchronously.

**notify_all_name(Name, Pid1, Pid2) -> none**

Types:
- `Name` = term()
- `Pid1` = `Pid2` = pid()

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

**random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2**

Types:
- `Name` = term()
- `Pid1` = `Pid2` = pid()

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

**random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2**

Types:
- `Name` = term()
- `Pid1` = `Pid2` = pid()

This function can be used as a name resolving function for `register_name/3` and `re_register_name/3`. It randomly chooses one of the pids for registration, and sends the message `{global_name_conflict, Name}` to the other pid.
register_name(Name, Pid)
register_name(Name, Pid, Resolve) -> yes | no

Types:
- Name = term()
- Pid = pid()
- Resolve = fun() or {Module, Function} where
  Resolve(Name, Pid, Pid2) -> Pid | Pid2 | none

Globally associates the name Name with a pid, that is, Globally notifies all nodes of a new global name in a network of Erlang nodes. When new nodes are added to the network, they are informed of the globally registered names that already exist. The network is also informed of any global names in newly connected nodes. If any name clashes are discovered, the Resolve function is called. Its purpose is to decide which pid is correct. If the function crashes, or returns anything other than one of the pids, the name is unregistered. This function is called once for each name clash.

There are three pre-defined resolve functions: random_exit_name/3, random_notify_name/3, and notify_all_name/3. If no Resolve function is defined, random_exit_name is used. This means that one of the two registered processes will be selected as correct while the other is killed.

This function is completely synchronous. This means that when this function returns, the name is either registered on all nodes or none.

The function returns yes if successful, no if it fails. For example, no is returned if an attempt is made to register an already registered process or to register a process with a name that is already in use.

Note:
Releases up to and including OTP R10 did not check if the process was already registered. As a consequence the global name table could become inconsistent. The old (buggy) behavior can be chosen by giving the Kernel application variable global_multi_name_action the value allow.

If a process with a registered name dies, or the node goes down, the name is unregistered on all nodes.

registered_names() -> [Name]

Types:
- Name = term()

Returns a lists of all globally registered names.

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

Types:
- Name = term()
- Pid = pid()
Kernel Reference Manual

- Resolve = fun() or {Module, Function} where
- Resolve(Name, Pid, Pid2) -> Pid | Pid2 | none

Atomically changes the registered name Name on all nodes to refer to Pid.

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

send(Name, Msg) -> Pid

Types:
- Name = term()
- Msg = term()
- Pid = pid()

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

Failure: If Name is not a globally registered name, the calling function will exit with reason {badarg, [Name, Msg]}.

set_lock(Id)
set_lock(Id, Nodes)
set_lock(Id, Nodes, Retries) -> boolean()

Types:
- Id = {ResourceId, LockRequesterId}
- ResourceId = term()
- LockRequesterId = term()
- Nodes = [node()]
- Retries = int() > = 0 | infinity

Sets a lock on the specified nodes (or on all nodes if none are specified) on ResourceId for LockRequesterId. If a lock already exists on ResourceId for another requester than LockRequesterId, and Retries is not equal to 0, the process sleeps for a while and will try to execute the action later. When Retries attempts have been made, false is returned, otherwise true. If Retries is infinity, true is eventually returned (unless the lock is never released).

If no value for Retries is given, infinity is used.

This function is completely synchronous.

If a process which holds a lock dies, or the node goes down, the locks held by the process are deleted.

The global name server keeps track of all processes sharing the same lock, that is, if two processes set the same lock, both processes must delete the lock.

This function does not address the problem of a deadlock. A deadlock can never occur as long as processes only lock one resource at a time. But if some processes try to lock two or more resources, a deadlock may occur. It is up to the application to detect and rectify a deadlock.

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

Synchronizes the global name server with all nodes known to this node. These are the nodes which are returned from erlang:nodes(). When this function returns, the global name server will receive global information from all nodes. This function can be called when new nodes are added to the network.

trans(Id, Fun)
trans(Id, Fun, Nodes)
trans(Id, Fun, Nodes, Retries) -> Res | aborted

Types:
- Id = {ResourceId, LockRequesterId}
- Res = [node()]
- Fun = fun() | {M, F}
- Nodes = [node()]
- Retries = int() > = 0 | infinity
- Res = term()

Sets a lock on Id using set_lock/3. If this succeeds, Fun() is evaluated and the result Res is returned. Returns aborted if the lock attempt failed. If Retries is set to infinity, the transaction will not abort. infinity is the default setting and will be used if no value is given for Retries.

unregister_name(Name) -> void()

Types:
- Name = term()

Removes the globally registered name Name from the network of Erlang nodes.

whereis_name(Name) -> pid() | undefined

Types:
- Name = term()

Returns the pid with the globally registered name Name. Returns undefined if the name is not globally registered.

See Also

global_group(3) [page 181], net_kernel(3) [page 204],
global_group

Erlang Module

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

The main advantage of dividing systems to global groups is that the background load decreases while the number of nodes to be updated is reduced when manipulating globally registered names.

The Kernel configuration parameter global_groups defines the global groups (see also kernel(6) [page 30], config(4) [page 245]:

{global_groups, [GroupTuple]}

Types:

- GroupTuple = {GroupName, [Node]} | {GroupName, PublishType, [Node]}
- GroupName = atom() (naming a global group)
- PublishType = normal | hidden
- Node = atom() (naming a node)

A GroupTuple without PublishType is the same as a GroupTuple with PublishType == normal.

A node started with the command line flag -hidden, see [erl(1)], is said to be a hidden node. A hidden node will establish hidden connections to nodes not part of the same global group, but normal (visible) connections to nodes part of the same global group.

A global group defined with PublishType == hidden, is said to be a hidden global group. All nodes in a hidden global group are hidden nodes, regardless if they are started with the -hidden command line flag or not.

For the processes and nodes to run smoothly using the global group functionality, the following criteria must be met:

- An instance of the global group server, global_group, must be running on each node. The processes are automatically started and synchronized when a node is started.
- All involved nodes must agree on the global group definition, or the behavior of the system is undefined.
- All nodes in the system should belong to exactly one global group.

In the following description, a group node is a node belonging to the same global group as the local node.
Exports

global_groups() -> {GroupName, GroupNames} | undefined

Types:
- GroupName = atom()
- GroupNames = [GroupName]

Returns a tuple containing the name of the global group the local node belongs to, and the list of all other known group names. Returns undefined if no global groups are defined.

info() -> [[Item, Info]]

Types:
- Item, Info - see below

Returns a list containing information about the global groups. Each element of the list is a tuple. The order of the tuples is not defined.

- {state, State} If the local node is part of a global group, State == synced. If no global groups are defined, State == no_conf.
- {own_group_name, GroupName} The name (atom) of the group that the local node belongs to.
- {own_group_nodes, Nodes} A list of node names (atoms), the group nodes.
- {synced_nodes, Nodes} A list of node names, the group nodes currently synchronized with the local node.
- {sync_error, Nodes} A list of node names, the group nodes with which the local node has failed to synchronize.
- {no_contact, Nodes} A list of node names, the group nodes to which there are currently no connections.
- {other_groups, Groups} Groups is a list of tuples {GroupName, Nodes}, specifying the name and nodes of the other global groups.
- {monitoring, Pids} A list of pids, specifying the processes which have subscribed to nodeup and nodedown messages.

monitor_nodes(Flag) -> ok

Types:
- Flag = bool()

Depending on Flag, the calling process starts subscribing (Flag == true) or stops subscribing (Flag == false) to node status change messages. A process which has subscribed will receive the messages {nodeup, Node} and {nodedown, Node} when a group node connects or disconnects, respectively.

own_nodes() -> Nodes

Types:
- Nodes = [Node]
- Node = node()
Returns the names of all group nodes, regardless of their current status.

registered_names(Where) -> Names

Types:
- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Names = [Name]
- Name = atom()

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

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

Types:
- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Name = atom()
- Msg = term()

Searches for Name, globally registered on the specified node or in the specified global group, or - if the Where argument is not provided - in any global group. The global groups are searched in the order in which they appear in the value of the global_groups configuration parameter.

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

sync() -> ok

Synchronizes the group nodes, that is, the global name servers on the group nodes. Also check the names globally registered in the current global group and unregisters them on any known node not part of the group.

If synchronization is not possible, an error report is sent to the error logger (see also error_logger(3)).

Failure: {error, {'invalid global_groups definition', Bad}} if the global_groups configuration parameter has an invalid value Bad.

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

Types:
- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Name = atom()
Searches for Name, globally registered on the specified node or in the specified global group, or – if the Where argument is not provided – in any global group. The global groups are searched in the order in which they appear in the value of the global groups configuration parameter.

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

NOTE

In the situation where a node has lost its connections to other nodes in its global group, but has connections to nodes in other global groups, a request from another global group may produce an incorrect or misleading result. For example, the isolated node may not have accurate information about registered names in its global group.

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

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

SEE ALSO

[erl(1)], global(3) [page 176]
This module contains the interface to the `heart` process. `heart` sends periodic heartbeats to an external port program, which is also named `heart`. The purpose of the `heart` port program is to check that the Erlang runtime system it is supervising is still running. If the `heart` port program has not received any heartbeats within `HEART_BEAT_TIMEOUT` seconds (default is 60 seconds), the system can be rebooted. Also, if the system is equipped with a hardware watchdog timer and is running Solaris, the watchdog can be used to supervise the entire system.

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

    % erl -heart ...

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

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

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

The `HEART_BEAT_TIMEOUT` and `HEART_BEAT_BOOT_DELAY` environment variables can be used to configure the heart timeouts; they can be set in the operating system shell before Erlang is started or be specified at the command line:

    % erl -heart -env HEART_BEAT_TIMEOUT 30 ...

The value (in seconds) must be in the range $10 < X <= 65535$.

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

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

```erlang
set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}

Types:
- Cmd = string()

Sets a temporary reboot command. This command is used if a HEART_COMMAND other
than the one specified with the environment variable should be used in order to reboot
the system. The new Erlang runtime system will (if it misbehaves) use the environment
variable HEART_COMMAND to reboot.

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

clear_cmd() -> ok

Clears the temporary boot command. If the system terminates, the normal
HEART_COMMAND is used to reboot.

```erlang
get_cmd() -> {ok, Cmd}

Types:
- Cmd = string()

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

Erlang Module

Provides access to TCP/IP protocols.

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

Two Kernel configuration parameters affect the behaviour of all sockets opened on an Erlang node: `inet_default_connect_options` can contain a list of default options used for all sockets returned when doing `connect`, and `inet_default_listen_options` can contain a list of default options used when issuing a `listen` call. When `accept` is issued, the values of the listen socket options are inherited, why no such application variable is needed for `accept`.

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

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

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

DATA TYPES

```
#hostent{h_addr_list = [ip_address()] % list of addresses for this host
  h_addrtype = inet | inet6 % h_aliases = [hostname()] % list of aliases
  h_aliases = [hostname()] % h_length = int() % length of address in bytes
  h_length = int() % h_name = hostname() % official name for host
  h_name = hostname() %

The record is defined in the Kernel include file "inet.hrl"

Add the following directive to the module:
-include_lib("kernel/include/inet.hrl chronological").

hostname() = atom() | string()

ip_address() = [N1,N2,N3,N4] % IPv4
  | [K1,K2,K3,K4,K5,K6,K7,K8] % IPv6
  Ni = 0..255
  Ki = 0..65535

posix()
```

an atom which is named from the Posix error codes used in Unix, and in the runtime libraries of most C compilers

socket()

see gen_tcp(3), gen_udp(3)

Addresses as inputs to functions can be either a string or a tuple. For instance, the IP address 150.236.20.73 can be passed to gethostbyaddr/1 either as the string “150.236.20.73” or as the tuple {150, 236, 20, 73}.

IPv4 address examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>ip_address()</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td>{127,0,0,1}</td>
</tr>
<tr>
<td>192.168.42.2</td>
<td>{192,168,42,2}</td>
</tr>
</tbody>
</table>

IPv6 address examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>ip_address()</th>
</tr>
</thead>
<tbody>
<tr>
<td>::1</td>
<td>{0,0,0,0,0,0,0,1}</td>
</tr>
<tr>
<td>::192.168.42.2</td>
<td>{0,0,0,0,0,(192 bsl 8) bor 168,(42 bsl 8) bor 2}</td>
</tr>
<tr>
<td>FFFF::192.168.42.2</td>
<td>{16#FFFF,0,0,0,0,(192 bsl 8) bor 168,(42 bsl 8) bor 2}</td>
</tr>
<tr>
<td>3ffe:b80:1f8d:2:204:acff:fe17:bf38</td>
<td>{16#3ffe,16#b80,16#1f8d,16#2,16#204,16#acff,16#fe17,16#bf38}</td>
</tr>
<tr>
<td>fe80::204:acff:fe17:bf38</td>
<td>{16#fe80,0,0,0,16#204,16#acff,16#fe17,16#bf38}</td>
</tr>
</tbody>
</table>

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

1> inet_parse:address("192.168.42.2").
   ok,{192,168,42,2}]
2> inet_parse:address("FFFF::192.168.42.2").
   ok,{65535,0,0,0,0,49320,10754}]

Exports

close(Socket) -> ok

Types:

• Socket = socket()

Closes a socket of any type.

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

Types:

• Par, Val - see below

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

Types:
- Posix = posix()
Returns a diagnostic error string. See the section below for possible Posix values and the corresponding strings.

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

Types:
- Host = ip_address() | string() | atom()
- Family = inet | inet6
- Address = ip_address()
- posix() = term()
Returns the IP-address for Host as a tuple of integers. Host can be an IP-address, a single hostname or a fully qualified hostname.

getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}

Types:
- Host = ip_address() | string() | atom()
- Addresses = [ip_address()]
- Family = inet | inet6
Returns a list of all IP-addresses for Host. Host can be an IP-address, a single hostname or a fully qualified hostname.

gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}

Types:
- Address = string() | ip_address()
- Hostent = #hostent{}
Returns a hostent record given an address.

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

Types:
- Hostname = hostname()
- Hostent = #hostent{}
Returns a hostent record given a hostname.

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

Types:
- Hostname = hostname()
- Family = inet | inet6
- Hostent = #hostent{}
Returns a hostent record given a hostname, restricted to the given address family.

gethostname() -> {ok, Hostname} | {error, posix()}

Kernel Application (KERNEL)
inet Kernel Reference Manual

Types:

- Hostname = string()
Returns the local hostname. Will never fail.

getopts(Socket, Options) -> OptionValues

Types:

- Socket = term()
- Options = [Opt]
- OptionValues = [{Opt, Val}]

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

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

Types:

- Socket = socket()
- Address = ip_address()
- Port = int()

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

port(Socket) -> {ok, Port}

Types:

- Socket = socket()
- Port = int()

Returns the local port number for a socket.

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

Types:

- Socket = socket()
- Address = ip_address()
- Port = int()

Returns the local address and port number for a socket.

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

Types:

- Socket = term()
- Options = [{Opt, Val}]
- Opt, Val – see below

Sets one or more options for a socket. The following options are available:
{active, Boolean} If the value is true, which is the default, everything received from the socket will be sent as messages to the receiving process. If the value is false (passive mode), the process must explicitly receive incoming data by calling gen_tcp:recv/2,3 or gen_udp:recv/2,3 (depending on the type of socket).

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

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

{broadcast, Boolean} (UDP sockets) Enable/disable permission to send broadcasts.

{delay_send, Boolean} Normally, when an Erlang process sends to a socket, the driver will try to immediately send the data. If that fails, the driver will use any means available to queue up the message to be sent whenever the operating system says it can handle it. Setting {delay_send, true} will make all messages queue up. This makes the messages actually sent onto the network be larger but fewer.

The option actually affects the scheduling of send requests versus Erlang processes instead of changing any real property of the socket. Needless to say it is an implementation specific option. Default is false.

{dontroute, Boolean} Enable/disable routing bypass for outgoing messages.

{exit_on_close, Boolean} By default this option is set to true.

The only reason to set it to false is if you want to continue sending data to the socket after a close has been detected, for instance if the peer has used gen_tcp:shutdown/2 [page 172] to shutdown the write side.

{header, Size} This option is only meaningful if the binary option was specified when the socket was created. If the header option is specified, the first Size number bytes of data received from the socket will be elements of a list, and the rest of the data will be a binary given as the tail of the same list. If for example Size == 2, the data received will match [Byte1,Byte2|Binary].

{keepalive, Boolean} (TCP/IP sockets) Enables/disables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and an error message will be sent to the controlling process. Default disabled.

{nodelay, Boolean} (TCP/IP sockets) If Boolean == true, the TCP_NODELAY option is turned on for the socket, which means that even small amounts of data will be sent immediately.

{packet, PacketType} (TCP/IP sockets) Defines the type of packets to use for a socket. The following values are valid:

- **raw** | 0 No packaging is done.
- **1** | **2** | **4** Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of header can be one, two, or four bytes; the order of the bytes is big-endian. Each send operation will generate the header, and the header will be stripped off on each receive operation.
- **asn1** | **cdr** | **sunrm** | **fcgi** | **tpkt** | **line** These packet types only have effect on receiving. When sending a packet, it is the responsibility of the application to supply a correct header. On receiving, however, there will be
one message sent to the controlling process for each complete packet received, and, similarly, each call to gen_tcp:recv/2,3 returns one complete packet.
The header is not stripped off.
The meanings of the packet types are as follows:

- asn1 - ASN.1 BER,
- sunrm - Sun's RPC encoding,
- cdr - CORBA (GIOP 1.1),
- fci - Fast CGI,
- tpkt - TPKT format [RFC1006],
- line - Line mode, a packet is a line terminated with newline, lines longer than the receive buffer are truncated.

{packet_size, Integer} (TCP/IP sockets) Sets the max allowed length of the packet body. If the packet header indicates that the length of the packet is longer than the max allowed length, the packet is considered invalid. The same happens if the packet header is too big for the socket receive buffer.

{rebuf, Integer} Gives the size of the receive buffer to use for the socket.

{reuseaddr, Boolean} Allows or disallows local reuse of port numbers. By default, reuse is disallowed.

{sndbuf, Integer} Gives the size of the send buffer to use for the socket.

{priority, Integer} Sets the SO_PRIORITY socket level option on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

{tos, Integer} Sets IP_TOS IP level options on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

Note that the default options for TCP/IP sockets can be changed with the Kernel configuration parameters mentioned in the beginning of this document.

POSIX Error Codes

- e2big - argument list too long
- 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
- efont - bad font file format
- ebusy - file busy
- echild - no children
- echrng - channel number out of range
- ecomm - communication error on send
- econnaborted - software caused connection abort
- econnrefused - connection refused
- econnreset - connection reset by peer
- edeadlk - resource deadlock avoided
- edeadlock - resource deadlock avoided
- edestaddrreq - destination address required
- edirty - mounting a dirty fs w/o force
- edom - math argument out of range
- edotdot - cross mount point
- edquot - disk quota exceeded
- eduppkg - duplicate package name
- eexist - file already exists
-efault - bad address in system call argument
- efbig - file too large
- ehostdown - host is down
- ehostunreach - host is unreachable
- eidrm - identifier removed
- einit - initialization error
- einprogress - operation now in progress
- eintr - interrupted system call
- einval - invalid argument
- eio - I/O error
- eisconn - socket is already connected
- eisdir - illegal operation on a directory
- eisnam - is a named file
- el2hlt - level 2 halted
- el2nsync - level 2 not synchronized
- el3hlt - level 3 halted
- el3rst - level 3 reset
- elbin - ELBIN
- elibacc - cannot access a needed shared library
- elibbad - accessing a corrupted shared library
- elibexec - cannot exec a shared library directly
- elibmax - attempting to link in more shared libraries than system limit
- elibscn - .lib section in a.out corrupted
- elnrng - link number out of range
- eloop - too many levels of symbolic links
- emfile - too many open files
- emlink - too many links
- emsgsize - message too long
- emultihop - multihop attempted
- enametoolong - file name too long
- enavail - not available
- enet - ENET
- enetdown - network is down
- enetreset - network dropped connection on reset
- enetunreach - network is unreachable
- enfile - file table overflow
- enoano - anode table overflow
- enobufs - no buffer space available
- enocsi - no CSI structure available
- enodata - no data available
- enodev - no such device
- enoent - no such file or directory
- enoexec - exec format error
- enolck - no locks available
- enolink - link has be severed
- enomem - not enough memory
- enomsg - no message of desired type
- enonet - machine is not on the network
- enopkg - package not installed
- enoprotoopt - bad protocol option
- enospc - no space left on device
- enosr - out of stream resources or not a stream device
- enosym - unresolved symbol name
- enosys - function not implemented
- enotblk - block device required
- enotconn - socket is not connected
- enotdir - not a directory
- enotempty - directory not empty
- enotnam - not a named file
- enotsock - socket operation on non-socket
- enotsup - operation not supported
- enotty - inappropriate device for ioctl
- enotuniq - name not unique on network
- enxio - no such device or address
- eopnonsupp - operation not supported on socket
- eperm - not owner
- epfnosupport - protocol family not supported
- epipe - broken pipe
- eproclim - too many processes
- eprocnosupport - bad procedure for program
- eprogmismatch - program version wrong
- eprogunavail - RPC program not available
- eproto - protocol error
- eprotoonosupport - protocol not supported
- eprotoprototype - 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 - EREMOTELEASE
- 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
- euclidean - structure needs cleaning
- eunatch - protocol driver not attached
- eusers - too many users
- eversion - version mismatch
- ewouldblock - operation would block
- exdev - cross-domain link
- exfull - message tables full
- nxdomain - the hostname or domain name could not be found
init

Erlang Module

The `init` module is pre-loaded and contains the code for the `init` system process which coordinates the start-up of the system. The first function evaluated at start-up is `boot(BootArgs)`, where `BootArgs` is a list of command line arguments supplied to the Erlang runtime system from the local operating system. See \[erl(1)\]. `init` reads the boot script which contains instructions on how to initiate the system. See \[script(4)\] for more information about boot scripts. `init` also contains functions to restart, reboot, and stop the system.

Exports

```erlang
boot(BootArgs) -> void()
```

Types:
- `BootArgs = [binary()]`

Starts the Erlang runtime system. This function is called when the emulator is started and coordinates system start-up.

- `BootArgs` are all command line arguments except the emulator flags, that is, flags and plain arguments. See \[erl(1)\].

`init` itself interprets some of the flags, see Command Line Flags [page 199] below. The remaining flags (“user flags”) and plain arguments are passed to the `init` loop and can be retrieved by calling `get_arguments/0` and `get_plain_arguments/0`, respectively.

```erlang
get_args() -> [Arg]
```

Types:
- `Arg = atom()`

Returns any plain command line arguments as a list of atoms (possibly empty). It is recommended that `get_plain_arguments/1` is used instead, because of the limited length of atoms.

```erlang
get_argument(Flag) -> {ok, Arg} | error
```

Types:
- `Flag = atom()`
- `Arg = [Values]`
- `Values = [string()]`

Returns all values associated with the command line user flag `Flag`. If `Flag` is provided several times, each `Values` is returned in preserved order.
There are also a number of flags, which are defined automatically and can be retrieved using this function:

**root** The installation directory of Erlang/OTP, $\$ROOT$.

```
2> init:get_argument(root).
{ok,[["/usr/local/otp/releases/otp_beam_solaris8_r10b_patched"]]]
```

**prognome** The name of the program which started Erlang.

```
3> init:get_argument(prognome).
{ok,[["erl"]]}
```

**home** The home directory.

```
4> init:get_argument(home).
{ok,[["/home/harry"]]]
```

Returns _error_ if there is no value associated with Flag.

**get_arguments()** -> Flags

Types:
- Flags = [{Flag, Values}]
- Flag = atom()
- Values = [string()]

Returns all command line flags, as well as the system defined flags, see _get_argument/1_.

**get_plain_arguments()** -> [Arg]

Types:
- Arg = string()

Returns any plain command line arguments as a list of strings (possibly empty).

**get_status()** -> {InternalStatus, ProvidedStatus}

Types:
- InternalStatus = starting | started | stopping
- ProvidedStatus = term()
The current status of the \texttt{init} process can be inspected. During system startup (initialization), \texttt{InternalStatus} is starting, and \texttt{ProvidedStatus} indicates how far the boot script has been interpreted. Each \{progress, Info\} term interpreted in the boot script affects \texttt{ProvidedStatus}, that is, \texttt{ProvidedStatus} gets the value of \texttt{Info}.

\begin{verbatim}
reboot() \rightarrow \texttt{void()}

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the \texttt{-heart} command line flag was given, the \texttt{heart} program will try to reboot the system. Refer to \texttt{heart(3)} for more information.

To limit the shutdown time, the time \texttt{init} is allowed to spend taking down applications, the \texttt{-shutdown_time} command line flag should be used.
\end{verbatim}

\begin{verbatim}
restart() \rightarrow \texttt{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 \texttt{BootArgs} are used again.

To limit the shutdown time, the time \texttt{init} is allowed to spend taking down applications, the \texttt{-shutdown_time} command line flag should be used.
\end{verbatim}

\begin{verbatim}
script_id() \rightarrow \texttt{Id}

Types:
\begin{itemize}
\item \texttt{Id = \text{term()}}
\end{itemize}

Get the identity of the boot script used to boot the system. \texttt{Id} can be any Erlang term. In the delivered boot scripts, \texttt{Id} is \{\texttt{Name}, \texttt{Vsn}\}. \texttt{Name} and \texttt{Vsn} are strings.
\end{verbatim}

\begin{verbatim}
stop() \rightarrow \texttt{void()}

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the \texttt{-heart} command line flag was given, the \texttt{heart} program is terminated before the Erlang node terminates. Refer to \texttt{heart(3)} for more information.

To limit the shutdown time, the time \texttt{init} is allowed to spend taking down applications, the \texttt{-shutdown_time} command line flag should be used.
\end{verbatim}
Command Line Flags

The \texttt{init} module interprets the following command line flags:

\begin{itemize}
\item [--] Everything following -- up to the next flag is considered plain arguments and can be retrieved using \texttt{get\_plain\_arguments/0}.
\item [-eval \texttt{Expr}] Scans, parses and evaluates an arbitrary expression \texttt{Expr} during system initialization. If any of these steps fail (syntax error, parse error or exception during evaluation), Erlang stops with an error message. Here is an example that seeds the random number generator:

\begin{verbatim}
\% erl -eval '{X,Y,Z}' = now(), random:seed(X,Y,Z).
\end{verbatim}

This example uses Erlang as a hexadecimal calculator:

\begin{verbatim}
\% erl -noshell -eval 'R = 16#1F+16#A0, io:format("\~.16B\n", [R])' \ 
-s erlang halt
\end{verbatim}

If multiple -eval expressions are specified, they are evaluated sequentially in the order specified. -eval expressions are evaluated sequentially with -s and -run function calls (this also in the order specified). As with -s and -run, an evaluation that does not terminate, blocks the system initialization process.

\item [-extra] Everything following -extra is considered plain arguments and can be retrieved using \texttt{get\_plain\_arguments/0}.
\item [-run \texttt{Mod \[Func \[Arg1, Arg2, ...\]]}] Evaluates the specified function call during system initialization. Func defaults to start. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list \texttt{[Arg1,Arg2,...]} as argument. All arguments are passed as strings. If an exception is raised, Erlang stops with an error message. Example:

\begin{verbatim}
\% erl -run foo -run foo bar -run foo bar baz 1 2
\end{verbatim}

This starts the Erlang runtime system and evaluates the following functions:

\begin{verbatim}
foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).
\end{verbatim}

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a -run call which does not return will block further processing; to avoid this, use some variant of spawn in such cases.
\end{itemize}
-s Mod [Func [Arg1, Arg2, ...]] Evaluates the specified function call during system initialization. Func defaults to start. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list [Arg1, Arg2, ...] as argument. All arguments are passed as atoms. If an exception is raised, Erlang stops with an error message.

Example:

```
% erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a -s call which does not return will block further processing; to avoid this, use some variant of spawn in such cases.

Due to the limited length of atoms, it is recommended that -run be used instead.

**Example**

```
% erl --a b -children thomas claire -ages 7 3 -- x y
...```

```
1> init:get_plain_arguments().
["a", "b", "x", "y"]
2> init:get_argument(children).
{ok, [["thomas", "claire"]]}  
3> init:get_argument(ages). 
{ok, [["7", "3"]]}   
4> init:get_argument(silly).  
error  
```

**SEE ALSO**

erl_prim_loader(3) [page 71], heart(3) [page 185]
net_adm

Erlang Module

This module contains various network utility functions.

Exports

dns_hostname(Host) -> {ok, Name} | {error, Host}

Types:
- Host = atom() | string()
- Name = string()

Returns the official name of Host, or {error, Host} if no such name is found. See also inet(3).

host_file() -> Hosts | {error, Reason}

Types:
- Hosts = [Host]
- Host = atom()
- Reason = term()

Reads the .hosts.erlang file, see the section Files below. Returns the hosts in this file as a list, or returns {error, Reason} if the file could not be read. See file(3) for possible values of Reason.

localhost() -> Name

Types:
- Name = string()

Returns the name of the local host. If Erlang was started with the -name command line flag, Name is the fully qualified name.

names() -> {ok, [{Name, Port}]} | {error, Reason}

names(Host) -> {ok, [{Name, Port}]} | {error, Reason}

Types:
- Name = string()
- Port = int()
- Reason = address | term()
Similar to `epmd -names`, see `epmd(1)`. Host defaults to the local host. Returns the names and associated port numbers of the Erlang nodes that `epmd` at the specified host has registered.

Returns `{error, address}` if `epmd` is not running. See `inet(3)` for other possible values of `Reason`.

```erlang
(arne@dunn)1> net_adm:names().
{ok, [{'arne', 40262}]}.
```

**ping(Node)**

- Types:
  - `Node = node()`
- Tries to set up a connection to `Node`. Returns `pang` if it fails, or `pong` if it is successful.

**world()**

- `world() -> [node()]`
- `world(Arg) -> [node()]`

  - Types:
    - `Arg = silent | verbose`
    - This function calls `names(Host)` for all hosts which are specified in the Erlang host file `.hosts.erlang`, collects the replies and then evaluates `ping(Node)` on all those nodes.
    - Returns the list of all found nodes, regardless of the return value of `ping(Node)`.
    - `Arg` defaults to `silent`. If `Arg == verbose`, the function writes information about which nodes it is pinging to `stdout`.
    - This function can be useful when a node is started, and the names of the other nodes in the network are not initially known.
    - Failure: `{error, Reason}` if `host_file()` returns `{error, Reason}`.

**world_list(Hosts)**

- `world_list(Hosts) -> [node()]`
- `world_list(Hosts, Arg) -> [node()]`

  - Types:
    - `Hosts = [Host]`
    - `Host = atom()`
    - `Arg = silent | verbose`
    - As `world/0,1`, but the hosts are given as argument instead of being read from `.hosts.erlang`. 
Files

The `.hosts.erlang` file consists of a number of host names written as Erlang terms. It is looked for in the current work directory, the user’s home directory, and `$OTP_ROOT` (the root directory of Erlang/OTP), in that order.

The format of the `.hosts.erlang` file must be one host name per line. The host names must be within quotes as shown in the following example:

'`super.eua.ericsson.se'`.  
'`renat.eua.ericsson.se'`.  
'`grouse.eua.ericsson.se'`.  
'`gauffin1.eua.ericsson.se'`.  

```
   ^ (new line)
```
The net kernel is a system process, registered as `net_kernel`, which must be running for
distributed Erlang to work. The purpose of this process is to implement parts of the
BIFs `spawn/4` and `spawn_link/4`, and to provide monitoring of the network.

An Erlang node is started using the command line flag `-name` or `-sname`:

```bash
$ erl -sname foobar
```

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

```erlang
1> net_kernel:start([foobar, shortnames]).
{ok,<0.64.0>}
```

If the node is started with the command line flag `-sname`, the node name will be
`foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain
name). If started with the `-name` flag, `Host` is the fully qualified domain name. See
erl(1).

Normally, connections are established automatically when another node is referenced.
This functionality can be disabled by setting the Kernel configuration parameter
`dist_auto_connect` to false, see kernel(6) [page 30]. In this case, connections must be
established explicitly by calling `net_kernel:connect_node/1`.

Which nodes are allowed to communicate with each other is handled by the magic

Exports

```erlang
alow(Nodes) -> ok | error
```

**Types:**
- `Nodes = [node()]`

Limits access to the specified set of nodes. Any access attempts made from (or to) nodes
not in `Nodes` will be rejected.

Returns `error` if any element in `Nodes` is not an atom.

```erlang
comnect_node(Node) -> true | false | ignored
```

**Types:**
- `Node = node()`
Establishes a connection to Node. Returns true if successful, false if not, and ignored if the local node is not alive.

```erlang
monitor_nodes(Flag) -> ok | ignored | Error
monitor_nodes(Flag, Options) -> ok | ignored | Error
```

Types:
- Flag = true | false
- Options = [Option]
- Option = see below
- Error = error | {error, term()}

The calling process subscribes or unsubscribes to node status change messages. A nodeup message is delivered to all subscribing process when a new node is connected, and a nodedown message is delivered when a node is disconnected.

If Flag is true, a new subscription is started. If Flag is false, all previous subscriptions - started with the same Options - are stopped. Two option lists are considered the same if they contain the same set of options.

The format of the node status change messages depends on Options. If Options is [], which is the default, the format is:

```
{nodeup, Node} | {nodedown, Node}
Node = node()
```

If Options /= [], the format is:

```
{nodeup, Node, InfoList} | {nodedown, Node, InfoList}
Node = node()
InfoList = [{Tag, Val}]
```

InfoList is a list of tuples. Its contents depends on Options, see below.

Also, when OptionList == [] only visible nodes, that is, nodes that appear in the result of nodes/0 [page 101], are monitored.

Option can be any of the following:

```
{node_type, NodeType} Currently valid values for NodeType are:
visible Subscribe to node status change messages for visible nodes only. The tuple {node_type, visible} is included in InfoList.
hidden Subscribe to node status change messages for hidden nodes only. The tuple {node_type, hidden} is included in InfoList.
all Subscribe to node status change messages for both visible and hidden nodes. The tuple {node_type, visible | hidden} is included in InfoList.
```

nodedown_reason The tuple {nodedown_reason, Reason} is included in InfoList in nodedown messages. Reason can be:
- connection_setup_failed The connection setup failed (after nodeup messages had been sent).
- no_network No network available.
- net_kernel_terminated The net_kernel process terminated.
- shutdown Unspecified connection shutdown.
- connection_closed The connection was closed.
- disconnect The connection was disconnected (forced from the current node).
Net tick timeout

send_net_tick_failed Failed to send net tick over the connection.

get_status_failed Status information retrieval from the Port holding the connection failed.

get_net_ticktime() -> Res

Types:
- Res = NetTicktime | {ongoing_change_to, NetTicktime}
- NetTicktime = int()

Gets net_ticktime (see kernel(6) [page 30]).

Currently defined return values (Res):

NetTicktime net_ticktime is NetTicktime seconds
{ongoing_change_to, NetTicktime} net_kernel is currently changing net_ticktime to NetTicktime seconds.

set_net_ticktime(NetTicktime) -> Res

set_net_ticktime(NetTicktime, TransitionPeriod) -> Res

Types:
- NetTicktime = int() > 0
- TransitionPeriod = int() >= 0
- Res = unchanged | {change_initiated} | {ongoing_change_to, NewNetTicktime}
- NewNetTicktime = int() > 0

Sets net_ticktime (see kernel(6) [page 30]) to NetTicktime seconds.

TransitionPeriod defaults to 60.

Some definitions:

The minimum transition traffic interval (MTTI) minimum(NetTicktime, PreviousNetTicktime)*1000 div 4 milliseconds.

The transition period The time of the least number of consecutive MTTIs to cover TransitionPeriod seconds following the call to set_net_ticktime/2 (i.e. ((TransitionPeriod*1000 - 1) div MTTI + 1)*MTTI milliseconds).

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

Note:
The net_ticktime changes have to be initiated on all nodes in the network (with the same NetTicktime) before the end of any transition period on any node; otherwise, connections may erroneously be disconnected.

Returns one of the following:
unchanged net_ticktime already had the value of NetTicktime and was left unchanged.

change_initiated net_kernel has initiated the change of net_ticktime to NetTicktime seconds

{ongoing_change_to, NewNetTicktime} The request was ignored; because, net_kernel was busy changing net_ticktime to NewNetTicktime seconds.

start([Name]) -> {ok, pid()} | {error, Reason}
start([Name, NameType]) -> {ok, pid()} | {error, Reason}
start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}

Types:
  • Name = atom()
  • NameType = shortnames | longnames
  • Reason = {already_started, pid()} | term() 

Note that the argument is a list with exactly one, two or three arguments. NameType defaults to longnames and Ticktime to 15000.

Turns a non-distributed node into a distributed node by starting net_kernel and other necessary processes.

stop() -> ok | {error, not_allowed | not_found}

Turns a distributed node into a non-distributed node. For other nodes in the network, this is the same as the node going down. Only possible when the net kernel was started using start/1, otherwise returns {error, not_allowed}. Returns {error, not_found} if the local node is not alive.
OS

Erlang Module

The functions in this module are operating system specific. Careless use of these functions will result in programs that will only run on a specific platform. On the other hand, with careful use these functions can be of help in enabling a program to run on most platforms.

Exports

**cmd(Command)** -> string()

Types:
- Command = string() | atom()

Executes Command in a command shell of the target OS, captures the standard output of the command and returns this result as a string. This function is a replacement of the previous unix:cmd/1; on a Unix platform they are equivalent.

Examples:

LsOut = os:cmd("ls"), % on unix platform
DirOut = os:cmd("dir"), % on Win32 platform

Note that in some cases, standard output of a command when called from another program (for example, os:cmd/1) may differ, compared to the standard output of the command when called directly from an OS command shell.

**find_executable(Name)** -> Filename | false

**find_executable(Name, Path)** -> Filename | false

Types:
- Name = string()
- Path = string()
- Filename = string()

These two functions look up an executable program given its name and a search path, in the same way as the underlying operating system. find_executable/1 uses the current execution path (that is, the environment variable PATH on Unix and Windows).

Path, if given, should conform to the syntax of execution paths on the operating system. The absolute filename of the executable program Name is returned, or false if the program was not found.

**getenv()** -> [string()]
Returns a list of all environment variables. Each environment variable is given as a single string on the format "VarName=Value", where VarName is the name of the variable and Value its value.

\[ \text{getenv(VarName)} \rightarrow \text{Value} \mid \text{false} \]

Types:
- \text{VarName} = \text{string()}
- \text{Value} = \text{string()}

Returns the Value of the environment variable VarName, or false if the environment variable is undefined.

\[ \text{getpid()} \rightarrow \text{Value} \]

Types:
- \text{Value} = \text{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.

\[ \text{putenv(VarName, Value)} \rightarrow \text{true} \]

Types:
- \text{VarName} = \text{string()}
- \text{Value} = \text{string()}

Sets a new Value for the environment variable VarName.

\[ \text{type()} \rightarrow \{\text{Osfamily}, \text{Osname}\} \mid \text{Osfamily} \]

Types:
- \text{Osfamily} = \text{win32} \mid \text{unix} \mid \text{vxworks}
- \text{Osname} = \text{atom()}

Returns the Osfamily and, in some cases, Osname of the current operating system. On Unix, Osname will have same value as \text{uname -s} returns, but in lower case. For example, on Solaris 1 and 2, it will be sunos.

In Windows, Osname will be either nt (on Windows NT), or windows (on Windows 95). On VxWorks the OS family alone is returned, that is vxworks.

\[ \text{version()} \rightarrow \{\text{Major}, \text{Minor}, \text{Release}\} \mid \text{VersionString} \]

\text{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.
Types:
- Major = Minor = Release = integer()
- VersionString = string()

Returns the operating system version. On most systems, this function returns a tuple, but a string will be returned instead if the system has versions which cannot be expressed as three numbers.

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

Erlang Module

Introduction

Packages are simply namespaces for modules. All old Erlang modules automatically belong to the top level ("empty-string") namespace, and do not need any changes.

The full name of a packaged module is written as e.g. "fee.fie.foe.foo", i.e., as atoms separated by periods, where the package name is the part up to but not including the last period; in this case "fee.fie.foe". A more concrete example is the module `erl.lang.term`, which is in the package `erl.lang`. Package names can have any number of segments, as in `erl.lang.list.sort`. The atoms in the name can be quoted, as in `foo.'Bar'.baz`, or even the whole name, as in `foo.bar.baz` but the concatenation of atoms and periods must not contain two consecutive period characters or end with a period, as in `foo..bar',foo.'bar', or foo.'bar`. The periods must not be followed by whitespace.

The code loader maps module names onto the file system directory structure. E.g., the module `erl.lang.term` corresponds to a file `.../erl/lang/term.beam` in the search path. Note that the name of the actual object file corresponds to the last part only of the full module name. (Thus, old existing modules such as `lists` simply map to `.../lists.beam`, exactly as before.)

A packaged module in a file "foo/bar/fred.erl" is declared as:

```erlang
-module(foo.bar.fred).
```

This can be compiled and loaded from the Erlang shell using `c(fred)`, if your current directory is the same as that of the file. The object file will be named `fred.beam`.

The Erlang search path works exactly as before, except that the package segments will be appended to each directory in the path in order to find the file. E.g., assume the path is `"/usr/lib/erl", "usr/local/lib/otp/legacy/ebin", "home/barney/erl"`. Then, the code for a module named `foo.bar.fred` will be searched for first as `"/usr/lib/erl/foo/bar/fred.beam"`, then `"/usr/local/lib/otp/legacy/ebin/foo/bar/fred.beam"` and `"/home/barney/erl/foo/bar/fred.beam"`. A module like `lists`, which is in the top-level package, will be looked for as `"/usr/lib/erl/lists.beam"`, `"usr/local/lib/otp/legacy/ebin/lists.beam"` and `"home/barney/erl/lists.beam"`.

Programming

Normally, if a call is made from one module to another, it is assumed that the called module belongs to the same package as the source module. The compiler automatically expands such calls. E.g., in:

```erlang
-module(foo.bar.ml).
-export([f/1]).

f(X) -> m2:g(X).
```

Kernel Application (KERNEL) 211
m2:g(X) becomes a call to foo.bar.m2 If this is not what was intended, the call can be written explicitly, as in

```prolog
-module(foo.bar.m1).
-export([f/1]).
```

```prolog
f(X) -> fee.fie.foe.m2:g(X).
```

Because the called module is given with an explicit package name, no expansion is done in this case.

If a module from another package is used repeatedly in a module, an import declaration can make life easier:

```prolog
-module(foo.bar.m1).
-export([f/1, g/1]).
-import(fee.fie.foe.m2).
```

```prolog
f(X) -> m2:g(X).
g(X) -> m2:h(X).
```

will make the calls to m2 refer to fee.fie.foe.m2. More generally, a declaration

```prolog
-import(Package.Module).
```

will cause calls to Module to be expanded to Package.Module.

Old-style function imports work as normal (but full module names must be used); e.g.:

```prolog
-import(fee.fie.foe.m2, [g/1, h/1]).
```

however, it is probably better to avoid this form of import altogether in new code, since it makes it hard to see what calls are really “remote”.

If it is necessary to call a module in the top-level package from within a named package, the module name can be written either with an initial period as in e.g. “.lists”, or with an empty initial atom, as in “’lists”. However, the best way is to use an import declaration - this is most obvious to the eye, and makes sure we don’t forget adding a period somewhere:

```prolog
-module(foo.bar.fred).
-export([f/1]).
-import(lists).
```

```prolog
f(X) -> lists:reverse(X).
```

The dot-syntax for module names can be used in any expression. All segments must be constant atoms, and the result must be a well-formed package/module name. E.g.:

```prolog
spawn(foo.bar.fred, f, [X])
```

is equivalent to

```prolog
spawn('foo.bar.fred', f, [X]).
```

The Erlang Shell

The shell also automatically expands remote calls, however currently no expansions are made by default. The user can change the behaviour by using the import/1 shell function (or its abbreviation use/1). E.g.:

```prolog
1> import(foo.bar.m).
ok
2> m:f().
```
will evaluate foo.bar.m:f(). If a new import is made of the same name, this overrides any previous import. (It is likely that in the future, some system packages will be pre-imported.)

In addition, the function \texttt{import\_all/1} (and its alias \texttt{use\_all/1}) imports all modules currently found in the path for a given package name. E.g., assuming the files “.../foo/bar/fred.beam”, “.../foo/bar/barney.beam” and “.../foo/bar/bambam.beam” can be found from our current path,

\begin{verbatim}
1> import_all(foo.bar).
\end{verbatim}

will make fred, barney and bambam expand to foo.bar.fred, foo.bar.barney and foo.bar.bambam, respectively.

Note: The compiler does not have an “import all” directive, for the reason that Erlang has no compile time type checking. E.g. if the wrong search path is used at compile time, a call \texttt{m:f(...) could be expanded to foo.bar.m:f(...)} without any warning, instead of the intended \texttt{frob.ozz.m:f(...)}, if package foo.bar happens to be found first in the path. Explicitly declaring each use of a module makes for safe code.

\section*{Exports}

\noindent no functions exported
This module implements process groups. The groups in this module differ from the groups in the module \texttt{pg} in several ways. In \texttt{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 \texttt{foobar}, there can be a set of processes (which can be located on different nodes) which are all members of the group \texttt{foobar}. There is no special functions for sending a message to the group. Instead, client functions should be written with the functions \texttt{get\_members/1} and \texttt{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 \	exttt{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.

\[\text{get}_\text{members} \text{(Name)} \rightarrow [\text{Pid}] \mid \{\text{error}, \text{Reason}\}\]

Types:
- \text{Name} = \text{term()}

Returns all processes in the group \text{Name}. This function should be used from within a client function that accesses the group. It is then optimized for speed.

\[\text{get}_\text{local}_\text{members} \text{(Name)} \rightarrow [\text{Pid}] \mid \{\text{error}, \text{Reason}\}\]

Types:
- \text{Name} = \text{term()}

Returns all processes running on the local node in the group \text{Name}. This function should to be used from within a client function that accesses the group. It is then optimized for speed.

\[\text{join(} \text{Name}, \text{Pid}) \rightarrow \text{ok} \mid \{\text{error}, \text{Reason}\}\]

Types:
- \text{Name} = \text{term()}

Joins the process \text{Pid} to the group \text{Name}.

\[\text{leave(} \text{Name}, \text{Pid}) \rightarrow \text{ok} \mid \{\text{error}, \text{Reason}\}\]

Types:
- \text{Name} = \text{term()}

Makes the process \text{Pid} leave the group \text{Name}.

\[\text{which}_\text{groups()} \rightarrow [\text{Name}]\]

Types:
- \text{Name} = \text{term()}

Returns a list of all known groups.

\[\text{start()}\]

\[\text{start}_\text{link()} \rightarrow \{\text{ok, Pid}\} \mid \{\text{error, Reason}\}\]

Starts the pg2 server. Normally, the server does not need to be started explicitly, as it is started dynamically if it is needed. This is useful during development, but in a target system the server should be started explicitly. Use configuration parameters for kernel for this.

See Also

kernel(6) [page 30], pg(3)
rpc

Erlang Module

This module contains services which are similar to remote procedure calls. It also contains broadcast facilities and parallel evaluators. A remote procedure call is a method to call a function on a remote node and collect the answer. It is used for collecting information on a remote node, or for running a function with some specific side effects on the remote node.

Exports

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

Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]
- Res = term()
- Reason = term()

Evaluates apply(Module, Function, Args) on the node Node and returns the corresponding value Res, or {badrpc, Reason} if the call fails.

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

Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]
- Res = term()
- Reason = timeout | term()
- Timeout = int() | infinity

Evaluates apply(Module, Function, Args) on the node Node and returns the corresponding value Res, or {badrpc, Reason} if the call fails. Timeout is a timeout value in milliseconds. If the call times out, Reason is timeout. If the reply arrives after the call times out, no message will contaminate the caller's message queue, since this function spawns off a middleman process to act as (a void) destination for such an orphan reply. This feature also makes this function more expensive than call/4 at the caller's end.

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

Types:
Kernel Reference Manual

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

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

\[
\text{block\_call(Node, Module, Function, Args, Timeout)} \rightarrow \text{Res | \{badrpc, Reason\}}
\]

Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]
- Timeout = int() | infinity
- Res = term()
- Reason = term()

Like block\_call/4, but with a timeout value in the same manner as call/5.

\[
\text{async\_call(Node, Module, Function, Args)} \rightarrow \text{Key}
\]

Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]
- Key – see below

Implements call streams with promises, a type of RPC which does not suspend the caller until the result is finished. Instead, a key is returned which can be used at a later stage to collect the value. The key can be viewed as a promise to deliver the answer.

In this case, the key Key is returned, which can be used in a subsequent call to yield/1 or nb\_yield/1,2 to retrieve the value of evaluating apply(Module, Function, Args) on the node Node.

\[
\text{yield(Key)} \rightarrow \text{Res | \{badrpc, Reason\}}
\]

Types:
- Key – see async\_call/4
- Res = term()
- Reason = term()

Returns the promised answer from a previous async\_call/4. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from Node.

\[
\text{nb\_yield(Key)} \rightarrow \\{\text{value, Val} \mid \text{timeout}\}
\]

Types:
- Key - see async_call/4
- Val = Res | {badrpc, Reason}
- Res = term()
- Reason = term()
Equivalent to nb_yield(Key, 0).

\[ nb\_yield(Key, \text{Timeout}) \rightarrow \{\text{value, Val} \mid \text{timeout} \} \]

Types:
- Key = see async_call/4
- Timeout = int() | infinity
- Val = Res | {badrpc, Reason}
- Res = term()
- Reason = term()
This is a non-blocking version of yield/1. It returns the tuple \{\text{value, Val} \} when the computation has finished, or \text{timeout} when \text{Timeout} milliseconds has elapsed.

\[ \text{multicall(Module, Function, Args)} \rightarrow \{\text{ResL, BadNodes} \} \]

Types:
- Module = Function = atom()
- Args = [term()]
- ResL = [term()]
- BadNodes = [node()]
Equivalent to multicall([node()|nodes()], Module, Function, Args, infinity).

\[ \text{multicall(Nodes, Module, Function, Args)} \rightarrow \{\text{ResL, BadNodes} \} \]

Types:
- Nodes = [node()]
- Module = Function = atom()
- Args = [term()]
- ResL = [term()]
- BadNodes = [node()]
Equivalent to multicall(Nodes, Module, Function, Args, infinity).

\[ \text{multicall(Module, Function, Args, Timeout)} \rightarrow \{\text{ResL, BadNodes} \} \]

Types:
- Module = Function = atom()
- Args = [term()]
- Timeout = int() | infinity
- ResL = [term()]
- BadNodes = [node()]
Equivalent to multicall([node()|nodes()], Module, Function, Args, Timeout).

\[ \text{multicall(Nodes, Module, Function, Args, Timeout)} \rightarrow \{\text{ResL, BadNodes} \} \]
In contrast to an RPC, a multicall is an RPC which is sent concurrently from one client to multiple servers. This is useful for collecting some information from a set of nodes, or for calling a function on a set of nodes to achieve some side effects. It is semantically the same as iteratively making a series of RPCs on all the nodes, but the multicall is faster as all the requests are sent at the same time and are collected one by one as they come back.

The function evaluates `apply(Module, Function, Args)` on the specified nodes and collects the answers. It returns `{ResL, Badnodes}`, where Badnodes is a list of the nodes that terminated or timed out during computation, and ResL is a list of the return values. Timeout is a time (integer) in milliseconds, or infinity.

The following example is useful when new object code is to be loaded on all nodes in the network, and also indicates some side effects RPCs may produce:

```prolog
%%% Find object code for module Mod
{Mod, Bin, File} = code:get_object_code(Mod),

%%% and load it on all nodes including this one
{ResL, _} = rpc:multicall(code, load_binary, [Mod, Bin, File,]),

%%% and then maybe check the ResL list.
```

\[\text{cast(Node, Module, Function, Args)} \rightarrow \text{void()}
\]

Types:
- \text{Node} = \text{node()}
- \text{Module} = \text{Function} = \text{atom()}
- \text{Args} = \text{[term()]}  

Evaluates `apply(Module, Function, Args)` on the node Node. No response is delivered and the calling process is not suspended until the evaluation is complete, as is the case with `call/4,5`.

\[\text{eval_everywhere(Module, Function, Args)} \rightarrow \text{void()}
\]

Types:
- \text{Module} = \text{Function} = \text{atom()}
- \text{Args} = \text{[term()]}  

Equivalent to `eval_everywhere([node()], Module, Function, Args)`.

\[\text{eval_everywhere(Nodes, Module, Function, Args)} \rightarrow \text{void()}
\]

Types:
- \text{Nodes} = \text{[node()]}  
- \text{Module} = \text{Function} = \text{atom()}

The function evaluates `apply(Module, Function, Args)` on the specified nodes and collects the answers. It returns `{ResL, Badnodes}`, where Badnodes is a list of the nodes that terminated or timed out during computation, and ResL is a list of the return values. Timeout is a time (integer) in milliseconds, or infinity.
• $\text{Args} = \text{[term()]}$

Evaluates $\text{apply(Module, Function, Args)}$ on the specified nodes. No answers are collected.

$\text{abcast(Name, Msg)} \Rightarrow \text{void()}$

Types:
• $\text{Name} = \text{atom()}$
• $\text{Msg} = \text{term()}$

Equivalent to $\text{abcast([node()|nodes()], Name, Msg)}$.

$\text{abcast(Nodes, Name, Msg)} \Rightarrow \text{void()}$

Types:
• $\text{Nodes} = \text{[node()]}$
• $\text{Name} = \text{atom()}$
• $\text{Msg} = \text{term()}$

Broadcasts the message $\text{Msg}$ asynchronously to the registered process $\text{Name}$ on the specified nodes.

$\text{sbcast(Name, Msg)} \Rightarrow \{\text{GoodNodes, BadNodes}\}$

Types:
• $\text{Name} = \text{atom()}$
• $\text{Msg} = \text{term()}$
• $\text{GoodNodes = BadNodes = [node()]}$

Equivalent to $\text{sbcast([node()|nodes()], Name, Msg)}$.

$\text{sbcast(Nodes, Name, Msg)} \Rightarrow \{\text{GoodNodes, BadNodes}\}$

Types:
• $\text{Nodes} = \text{[node()]}$
• $\text{Name} = \text{atom()}$
• $\text{Msg} = \text{term()}$
• $\text{GoodNodes = BadNodes = [node()]}$

Broadcasts the message $\text{Msg}$ synchronously to the registered process $\text{Name}$ on the specified nodes.

Returns $\{\text{GoodNodes, BadNodes}\}$, where $\text{GoodNodes}$ is the list of nodes which have $\text{Name}$ as a registered process.

The function is synchronous in the sense that it is known that all servers have received the message when the call returns. It is not possible to know that the servers have actually processed the message.

Any further messages sent to the servers, after this function has returned, will be received by all servers after this message.

$\text{server\_call(Node, Name, ReplyWrapper, Msg)} \Rightarrow \text{Reply | \{error, Reason\}}$

Types:
• $\text{Node} = \text{node()}$
• $\text{Name} = \text{atom()}$
• ReplyWrapper = Msg = Reply = term()
• Reason = term()

This function can be used when interacting with a server called \( \text{Name} \) at node \( \text{Node} \). It is assumed that the server receives messages in the format \{From, Msg\} and replies using From ! \{ReplyWrapper, Node, Reply\}. This function makes such a server call and ensures that the entire call is packed into an atomic transaction which either succeeds or fails. It never hangs, unless the server itself hangs.

The function returns the answer \( \text{Reply} \) as produced by the server \( \text{Name} \), or \{error, Reason\}.

\[
\text{multi\_server\_call}(\text{Name}, \text{Msg}) \rightarrow \{\text{Replies}, \text{BadNodes}\}
\]

Types:
• \( \text{Name} \) = atom()
• \( \text{Msg} \) = term()
• \( \text{Replies} \) = \([\text{Reply}]\)
• \( \text{Reply} \) = term()
• \( \text{BadNodes} \) = \([\text{node()}]\)

Equivalent to \text{multi\_server\_call([node()]|\text{nodes()}), Name, Msg)}.

\[
\text{multi\_server\_call}(\text{Nodes}, \text{Name}, \text{Msg}) \rightarrow \{\text{Replies}, \text{BadNodes}\}
\]

Types:
• \( \text{Nodes} \) = \([\text{node()}]\)
• \( \text{Name} \) = atom()
• \( \text{Msg} \) = term()
• \( \text{Replies} \) = \([\text{Reply}]\)
• \( \text{Reply} \) = term()
• \( \text{BadNodes} \) = \([\text{node()}]\)

This function can be used when interacting with servers called \( \text{Name} \) on the specified nodes. It is assumed that the servers receive messages in the format \{From, Msg\} and reply using From ! \{Name, Node, Reply\}, where \( \text{Node} \) is the name of the node where the server is located. The function returns \{Replies, Badnodes\}, where \( \text{Replies} \) is a list of all \( \text{Reply} \) values and \( \text{BadNodes} \) is a list of the nodes which did not exist, or where the server did not exist, or where the server terminated before sending any reply.

\[
\text{safe\_multi\_server\_call}(\text{Name}, \text{Msg}) \rightarrow \{\text{Replies}, \text{BadNodes}\}
\]

\[
\text{safe\_multi\_server\_call}(\text{Nodes}, \text{Name}, \text{Msg}) \rightarrow \{\text{Replies}, \text{BadNodes}\}
\]

\textbf{Warning:}
This function is deprecated. Use \text{multi\_server\_call(2,3)} instead.

In Erlang/OTP R6B and earlier releases, \text{multi\_server\_call(2,3)} could not handle the case where the remote node exists, but there is no server called \( \text{Name} \). Instead this function had to be used. In Erlang/OTP R7B and later releases, however, the functions are equivalent, except for this function being slightly slower.
parallel_eval(FuncCalls) -> ResL

Types:
- FuncCalls = [{Module, Function, Args}]
  - Module = Function = atom()
  - Args = [term()]
  - ResL = [term()]

For every tuple in FuncCalls, evaluates apply(Module, Function, Args) on some node in the network. Returns the list of return values, in the same order as in FuncCalls.

pmap([{Module, Function}], ExtraArgs, List2) -> List1

Types:
- Module = Function = atom()
- ExtraArgs = [term()]
- List1 = [Elem]
- Elem = term()
- List2 = [term()]

Evaluates apply(Module, Function, [Elem|ExtraArgs]), for every element Elem in List1, in parallel. Returns the list of return values, in the same order as in List1.

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

Types:
- Pid = pid()
- Item, Info - see erlang:process_info/1

Location transparent version of the BIF process_info/1.

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

Types:
- Pid = pid()
- Item, Info - see erlang:process_info/1

Location transparent version of the BIF process_info/2.
Sequential tracing makes it possible to trace all messages resulting from one initial message. Sequential tracing is completely independent of the ordinary tracing in Erlang, which is controlled by the `erlang:trace/3` BIF. See the chapter What is Sequential Tracing [page 226] 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**

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

  ```erlang
  OldToken = seq_trace:set_token([]), % set to empty and save % old value
  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.

  **Declaration:**
  ```erlang
  set_token(Token) -> PreviousToken
  ```

  **Types:**

  - `Token` = `PreviousToken` = `term()` | `[]`

  **Set token:**

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

  **Types:**

  - `Component` = `label` | `serial` | `Flag`
  - `Flag` = `send` | `'receive'` | `print` | `timestamp`
- \( \text{Val} = \text{OldVal} - \text{see below} \)

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

\[
\text{set} \_ \text{token}(\text{label}, \text{Int}) \quad \text{The label component is an integer which identifies all events belonging to the same sequential trace. If several sequential traces can be active simultaneously, label is used to identify the separate traces. Default is 0.}
\]

\[
\text{set} \_ \text{token}(\text{serial}, \text{SerialValue}) \quad \text{SerialValue} = \{\text{Previous, Current}\}. \text{The serial component contains counters which enables the traced messages to be sorted, should never be set explicitly by the user as these counters are updated automatically. Default is \{0, 0\}.}
\]

\[
\text{set} \_ \text{token}(\text{send}, \text{Bool}) \quad \text{A trace token flag (true | false) which enables/disables tracing on message sending. Default is false.}
\]

\[
\text{set} \_ \text{token}(\text{‘receive’}, \text{Bool}) \quad \text{A trace token flag (true | false) which enables/disables tracing on message reception. Default is false.}
\]

\[
\text{set} \_ \text{token}(\text{print}, \text{Bool}) \quad \text{A trace token flag (true | false) which enables/disables tracing on explicit calls to seq_trace:print/1. Default is false.}
\]

\[
\text{set} \_ \text{token}(\text{timestamp}, \text{Bool}) \quad \text{A trace token flag (true | false) which enables/disables a timestamp to be generated for each traced event. Default is false.}
\]

\[
\text{get} \_ \text{token}() \to \text{TraceToken}
\]

Types:
- \( \text{TraceToken} = \text{term()} | [] \)

Returns the value of the trace token for the calling process. If [] is returned, it means that tracing is not active. Any other value returned is the value of an active trace token. The value returned can be used as input to the set_token/1 function.

\[
\text{get} \_ \text{token}(\text{Component}) \to \{\text{Component}, \text{Val} \}
\]

Types:
- \( \text{Component} = \text{label} | \text{serial} | \text{Flag} \)
- \( \text{Flag} = \text{send} | \text{‘receive’} | \text{print} | \text{timestamp} \)
- \( \text{Val} = \text{see set_token/2} \)

Returns the value of the trace token component Component. See set_token/2 [page 223] for possible values of Component and Val.

\[
\text{print} \_ \{(\text{TraceInfo}) \to \text{void}() \}
\]

Types:
- \( \text{TraceInfo} = \text{term()} \)

Puts the Erlang term TraceInfo into the sequential trace output if the calling process currently is executing within a sequential trace and the print flag of the trace token is set.

\[
\text{print}(\text{Label}, \text{TraceInfo}) \to \text{void}() \]

Types:
Kernel Reference Manual

- Label = int()
- TraceInfo = term()

Same as print/1 with the additional condition that TraceInfo is output only if Label is equal to the label component of the trace token.

`reset_trace()` -> void()

Sets the trace token to empty for all processes on the local node. The process internal counters used to create the serial of the trace token is set to 0. The trace token is set to empty for all messages in message queues. Together this will effectively stop all ongoing sequential tracing in the local node.

`set_system_tracer(Tracer)` -> OldTracer

Types:
- Tracer = OldTracer = pid() | port() | false

Sets the system tracer. The system tracer can be either a process or port denoted by Tracer. Returns the previous value (which can be false if no system tracer is active).

Failure: ![badarg, Info] if Pid is not an existing local pid.

`get_system_tracer()` -> Tracer

Types:
- Tracer = pid() | port() | false

Returns the pid or port identifier of the current system tracer or false if no system tracer is activated.

Trace Messages Sent To the System Tracer

The format of the messages are:

{seq_trace, Label, SeqTraceInfo, TimeStamp}

or

{seq_trace, Label, SeqTraceInfo}

depending on whether the timestamp flag of the trace token is set to true or false. Where:

- Label = int()
- TimeStamp = {Seconds, Milliseconds, Microseconds}
  - Seconds = Milliseconds = Microseconds = int()

The SeqTraceInfo can have the following formats:

- {send, Serial, From, To, Message} Used when a process From with its trace token flag print set to true has sent a message.
- {‘receive’, Serial, From, To, Message} Used when a process To receives a message with a trace token that has the ‘receive’ flag set to true.
- {print, Serial, From, _, Info} Used when a process From has called `seq_trace:print(Label, TraceInfo)` and has a trace token with the print flag set to true and label set to Label.
Serial is a tuple \( \{ \text{PreviousSerial}, \text{ThisSerial} \} \), where the first integer 
\text{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, 
\text{PreviousSerial} is set to the value of the process internal "trace clock". The second integer \text{ThisSerial} is the serial counter that a process sets on outgoing messages and it is based on the process internal "trace clock" which is incremented by one before it is attached to the trace token in the message.

**What is Sequential Tracing**

Sequential tracing is a way to trace a sequence of messages sent between different local or remote processes, where the sequence is initiated by one single message. In short it works like this:

Each process has a trace token, which can be empty or not empty. When not empty the trace token can be seen as the tuple \( \{ \text{Label}, \text{Flags}, \text{Serial}, \text{From} \} \). The trace token is passed invisibly with each message.

In order to start a sequential trace the user must explicitly set the trace token in the process that will send the first message in a sequence.

The trace token of a process is set each time the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not. On each Erlang node a process can be set as the system tracer. This process will receive trace messages each time a message with a trace token is sent or received (if the trace token flag 'send' or 'receive' is set). The system tracer can then print each trace event, write it to a file or whatever suitable.

**Note:**
The system tracer will only receive those trace events that occur locally within the Erlang node. To get the whole picture of a sequential trace that involves processes on several Erlang nodes, the output from the system tracer on each involved node must be merged (off line).

In the following sections Sequential Tracing and its most fundamental concepts are described.

**Trace Token**

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

The current token of a process is set in two ways, either

1. explicitly by the process itself, through a call to `seq_trace:set_token`, or
2. when a message is received.
In both cases the current token will be set. In particular, if the token of a message received is empty, the current token of the process is set to empty.

A trace token contains a label, and a set of flags. Both the label and the flags are set in 1 and 2 above.

Serial

The trace token contains a component which is called serial. It consists of two integers Previous and Current. The purpose is to uniquely identify each traced event within a trace sequence and to order the messages chronologically and in the different branches if any.

The algorithm for updating Serial can be described as follows:

Let each process have two counters prev_cnt and curr_cnt which both are set to 0 when a process is created. The counters are updated at the following occasions:

- When the process is about to send a message and the trace token is not empty.
  
  ```
  curr_cnt := curr_cnt + 1
  tprev := prev_cnt
  tcurr := curr_cnt
  ```

  The trace token with tprev and tcurr is then passed along with the message.

- When the process calls `seq_trace:print(Label, Info)`, Label matches the label part of the trace token and the trace token print flag is true.
  
  The same algorithm as for send above.

- When a message is received and contains a nonempty trace token.
  
  ```
  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 \texttt{seq\_trace:}\texttt{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 \texttt{seq\_trace:}\texttt{set\_token/2}.

Distribution

Sequential tracing between nodes is performed transparently. This applies to C-nodes built with ErlInterface too. A C-node built with ErlInterface 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 \texttt{Pid == \langle 0.30.0 \rangle} like this:

\begin{verbatim}
-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).
\end{verbatim}

And a registered process \texttt{call\_server} with \texttt{Pid == \langle 0.31.0 \rangle} 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 AX E-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
    ~n", [To, Serial, From, Message]);
print_trace({send, Serial, From, To, Message}) ->
    io:format("p Sent ~p TO ~p WITH ~n
    ~n", [From, Serial, To, Message]).

The code that creates a process that runs the tracer function above and sets that process as the system tracer could look like this:
start() ->
    Pid = spawn(?MODULE, tracer, []),
    seq_trace:set_system_tracer(Pid), % set Pid as the system tracer
    ok.

With a function like test/0 below the whole example can be started.

test() ->
    P = spawn(?MODULE, loop, [port]),
    register(call_server, spawn(?MODULE, loop, [])),
    start(),
    P ! {port, message}.
user

Erlang Module

user is a server which responds to all the messages defined in the I/O interface. The code in user.erl can be used as a model for building alternative I/O servers.
wrap_log_reader

Erlang Module

wrap_log_reader is a function to read internally formatted wrap disk logs, refer to disk_log(3). wrap_log_reader does not interfere with disk_log activities; there is however a known bug in this version of the wrap_log_reader, see chapter bugs below.

A wrap disk log file consists of several files, called index files. A log file can be opened and closed. It is also possible to open just one index file separately. If an non-existent or a non-internally formatted file is opened, an error message is returned. If the file is corrupt, no attempt to repair it will be done but an error message is returned.

If a log is configured to be distributed, there is a possibility that all items are not loggen on all nodes. wrap_log_reader does only read the log on the called node, it is entirely up to the user to be sure that all items are read.

Exports

chunk(Continuation)
chunk(Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | {Continuation2, eof} | {error, Reason}

Types:
- Continuation = continuation()
- N = int() > 0 | infinity
- Continuation2 = continuation()
- Terms = [term()]
- Badbytes = integer()

This function makes it possible to efficiently read the terms which have been appended to a log. It minimises disk I/O by reading large 8K chunks from the file.

The first time chunk is called an initial continuation returned from the open/1, open/2 must be provided.

When chunk/3 is called, N controls the maximum number of terms that are read from the log in each chunk. Default is infinity, which means that all the terms contained in the 8K chunk are read. If less than N terms are returned, this does not necessarily mean that end of file is reached.

The chunk function returns a tuple {Continuation2, Terms}, where Terms is a list of terms found in the log. Continuation2 is yet another continuation which must be passed on into any subsequent calls to chunk. With a series of calls to chunk it is then possible to extract all terms from a log.

The chunk function returns a tuple {Continuation2, Terms, Badbytes} if the log is opened in read only mode and the read chunk is corrupt. Badbytes indicates the number of non-Erlang terms found in the chunk. Note also that the log is not repaired.
chunk returns \(\{\text{Continuation2}, \text{eof}\}\) when the end of the log is reached, and \(\{\text{error}, \text{Reason}\}\) if an error occurs.

The returned continuation may or may not be valid in the next call to \textit{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.

\textbf{close(Continuation) \rightarrow ok}

Types:
- \textit{Continuation} = \text{continuation()}

This function closes a log file properly.

\textbf{open(Filename) \rightarrow OpenRet}

\textbf{open(Filename, N) \rightarrow OpenRet}

Types:
- \textit{File} = \text{string()} \mid \text{atom()}
- \textit{N} = \text{integer()}
- \textit{OpenRet} = \{\text{ok, Continuation}\} \mid \{\text{error, Reason}\}
- \textit{Continuation} = \text{continuation()}

\textit{Filename} specifies the name of the file which is to be read.

\textit{N} specifies the index of the file which is to be read. If \textit{N} is omitted the whole wrap log file will be read; if it is specified only the specified index file will be read.

The \textit{open} function returns \(\{\text{ok, Continuation}\}\) if the log/index file was successfully opened. The \textit{Continuation} is to be used when chunking or closing the file.

The function returns \(\{\text{error, Reason}\}\) for all errors.

\textbf{Bugs}

This version of the \textit{wrap\_log\_reader} does not detect if the \textit{disk\_log} wraps to a new index file between a \textit{wrap\_log\_reader:open} and the first \textit{wrap\_log\_reader:chunk}. In this case the chuck will actually read the last logged items in the log file, because the opened index file was truncated by the \textit{disk\_log}.

\textbf{See Also}

disk\_log(3) [page 53]
The zlib module provides an API for the zlib library (http://www.zlib.org). It is used to compress and decompress data. The data format is described by RFCs 1950 to 1952. A typical (compress) usage looks like:

```
Z = zlib:open(),
ok = zlib:deflateInit(Z,default),
Compress = fun(end of data, _Cont) -> [];
    (Data, Cont) ->
      [zlib:deflate(Z, Data)|Cont(Read(),Cont)]
  end,
Compressed = Compress(Read(),Compress),
Last = zlib:deflate(Z, [], finish),
ok = zlib:deflateEnd(Z),
zlib:close(Z),
list_to_binary([Compressed|Last])
```

In all functions errors, `{EXIT,Reason,Backtrace}`, might be thrown, where `Reason` describes the error. Typical reasons are:

- **badarg** Bad argument
- **data_error** The data contains errors
- **stream_error** Inconsistent stream state
- **einval** Bad value or wrong function called
- `{need dictionary,Adler32}` See `inflate/2`

### DATA TYPES

- **iodata** = iolist() | binary()
  - iolist = [char() | binary() | iolist()]
    - a binary is allowed as the tail of the list
- **zstream** = a zlib stream, see open/0
Exports

open() -> Z

Types:
• Z = zstream()
Open a zlib stream.

close(Z) -> ok

Types:
• Z = zstream()
Closes the stream referenced by Z.

deflateInit(Z) -> ok

Types:
• Z = zstream()
Same as zlib:deflateInit(Z, default).

deflateInit(Z, Level) -> ok

Types:
• Z = zstream()
• Level = none | default | best_speed | best_compression | 0..9
Initialize a zlib stream for compression.
Level decides the compression level to be used, 0 (none), gives no compression at all, 1 (best_speed) gives best speed and 9 (best_compression) gives best compression.

deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok

Types:
• Z = zstream()
• Level = none | default | best_speed | best_compression | 0..9
• Method = deflated
• WindowBits = 9..15 | -9..-15
• MemLevel = 1..9
• Strategy = default | filtered | huffman_only
Initiates a zlib stream for compression.

The `Level` parameter decides the compression level to be used. 0 (`none`) gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression.

The `Method` parameter decides which compression method to use, currently the only supported method is `deflated`.

The `WindowBits` parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 9 through 15. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if `deflateInit/2`. A negative `WindowBits` value suppresses the zlib header (and checksum) from the stream. Note that the zlib source mentions this only as an undocumented feature.

The `MemLevel` parameter specifies how much memory should be allocated for the internal compression state. `MemLevel=1` uses minimum memory but is slow and reduces compression ratio; `MemLevel=9` uses maximum memory for optimal speed. The default value is 8.

The `Strategy` parameter is used to tune the compression algorithm. Use the value `default` for normal data, `filtered` for data produced by a filter (or predictor), or `huffman_only` to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of `filtered` is to force more Huffman coding and less string matching; it is somewhat intermediate between `default` and `huffman_only`. The `Strategy` parameter only affects the compression ratio but not the correctness of the compressed output even if it is not set appropriately.

```haskell
deflate(Z, Data) -> Compressed

Types:
  • Z = zstream()
  • Data = iodata()
  • Compressed = iolist()

Same as `deflate(Z, Data, none)`.
```

```haskell
deflate(Z, Data, Flush) ->

Types:
  • Z = zstream()
  • Data = iodata()
  • Flush = none | sync | full | finish
  • Compressed = iolist()
```

deflate/3 compresses as much data as possible, and stops when the input buffer becomes empty. It may introduce some output latency (reading input without producing any output) except when forced to flush.

If the parameter `Flush` is set to `sync`, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.
If **Flush** is set to **full**, all output is flushed as with **sync**, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using **full** too often can seriously degrade the compression.

If the parameter **Flush** is set to **finish**, pending input is processed, pending output is flushed and **deflate/3** returns. Afterwards the only possible operations on the stream are **deflateReset/1** or **deflateEnd/1**.

Flush can be set to **finish** immediately after **deflateInit** if all compression is to be done in one step.

```haskell
zlib:deflateInit(Z),
B1 = zlib:deflate(Z,Data),
B2 = zlib:deflate(Z,<><>,finish),
zlib:deflateEnd(Z),
list_to_binary([B1,B2]),
```

---

**deflateSetDictionary(Z, Dictionary) -> Adler32**

**Types:**
- **Z** = zstream()
- **Dictionary** = binary()
- **Adler32** = integer()

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after **deflateInit/[1|2|6]** or **deflateReset/1**, before any call of **deflate/3**. The compressor and decompressor must use exactly the same dictionary (see **inflateSetDictionary/2**). The adler checksum of the dictionary is returned.

**deflateReset(Z) -> ok**

**Types:**
- **Z** = zstream()

This function is equivalent to **deflateEnd/1** followed by **deflateInit/[1|2|6]**, but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes.

**deflateParams(Z, Level, Strategy) -> ok**

**Types:**
- **Z** = zstream()
- **Level** = none | default | best_speed | best_compression | 0..9
- **Strategy** = default | filtered | huffman_only
Dynamically update the compression level and compression strategy. The interpretation
of Level and Strategy is as in deflateInit/6. This can be used to switch between
compression and straight copy of the input data, or to switch to a different kind of input
data requiring a different strategy. If the compression level is changed, the input
available so far is compressed with the old level (and may be flushed); the new level will
take effect only at the next call of deflate/3.
Before the call of deflateParams, the stream state must be set as for a call of deflate/3,
since the currently available input may have to be compressed and flushed.

\[\text{deflateEnd}(Z) \rightarrow \text{ok}\]

Types:
- \(Z = \text{zstream}()\)

End the deflate session and cleans all data used. Note that this function will throw an
data\_error exception if the last call to deflate/3 was not called with Flush set to
finish.

\[\text{inflateInit}(Z) \rightarrow \text{ok}\]

Types:
- \(Z = \text{zstream}()\)

Initialize a zlib stream for decompression.

\[\text{inflateInit}(Z, \text{WindowBits}) \rightarrow \text{ok}\]

Types:
- \(Z = \text{zstream}()\)
- \(\text{WindowBits} = 9..15 | -9..-15\)

Initialize decompression session on zlib stream.

The WindowBits parameter is the base two logarithm of the maximum window size
(the size of the history buffer). It should be in the range 9 through 15. The default
value is 15 if inflateInit/1 is used. If a compressed stream with a larger window size
is given as input, inflate() will throw the data\_error exception. A negative WindowBits
value makes zlib ignore the zlib header (and checksum) from the stream. Note that the
zlib source mentions this only as an undocumented feature.

\[\text{inflate}(Z, \text{Data}) \rightarrow \text{DeCompressed}\]

Types:
- \(Z = \text{zstream}()\)
- \(\text{Data} = \text{iodata}()\)
- \(\text{DeCompressed} = \text{iolist}()\)

inflate/2 decompresses as much data as possible. It may some introduce some output
latency (reading input without producing any output).

If a preset dictionary is needed at this point (see inflateSetDictionary below),
inflate/2 throws a \{need\_dictionary, Adler\} exception where Adler is the adler32
checksum of the dictionary chosen by the compressor.

\[\text{inflateSetDictionary}(Z, \text{Dictionary}) \rightarrow \text{ok}\]
Types:
- \( Z = \text{zstream()} \)
- \( \text{Dictionary} = \text{binary()} \)

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call of `inflate/2` if this call threw a \{\text{need dictionary}, Adler\} exception. The dictionary chosen by the compressor can be determined from the Adler value thrown by the call to `inflate/2`. The compressor and decompressor must use exactly the same dictionary (see `deflateSetDictionary/2`).

Example:

```erlang
unpack(Z, Compressed, Dict) ->
    case catch zlib:inflate(Z, Compressed) of
        {'EXIT', [{need_dictionary, DictID}, _]} ->
            zlib:inflateSetDictionary(Z, Dict),
            Uncompressed = zlib:inflate(Z, []);
        Uncompressed ->
            Uncompressed
    end.
```

`inflateReset(Z) -> ok`

Types:
- \( Z = \text{zstream()} \)

This function is equivalent to `inflateEnd/1` followed by `inflateInit/1`, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by `inflateInit/[1|2]`.

`inflateEnd(Z) -> ok`

Types:
- \( Z = \text{zstream()} \)

End the inflate session and cleans all data used. Note that this function will throw a `data_error` exception if no end of stream was found (meaning that not all data has been uncompressed).

`setBufSize(Z, Size) -> ok`

Types:
- \( Z = \text{zstream()} \)
- \( \text{Size} = \text{integer()} \)

Sets the intermediate buffer size.

`getBufSize(Z) -> Size`

Types:
- \( Z = \text{zstream()} \)
- \( \text{Size} = \text{integer()} \)

Get the size of intermediate buffer.
crc32(Z) \rightarrow CRC

Types:
- Z = zstream()
- CRC = integer()

Get the current calculated CRC checksum.

crc32(Z, Binary) \rightarrow CRC

Types:
- Z = zstream()
- Binary = binary()
- CRC = integer()

Calculate the CRC checksum for Binary.

crc32(Z, PrevCRC, Binary) \rightarrow CRC

Types:
- Z = zstream()
- PrevCRC = integer()
- Binary = binary()
- CRC = integer()

Update a running CRC checksum for Binary. If Binary is the empty binary, this function returns the required initial value for the crc.

\[
\text{Crc} = \text{lists:foldl}(\text{fun}(\text{Bin}, \text{Crc0}) \rightarrow \\
\quad \text{zlib:crc32}(\text{Z}, \text{Crc0}, \text{Bin}), \\
\quad \text{end, zlib:crc32}(\text{Z}, <<>>, \text{Bins}),
\]

adler32(Z, Binary) \rightarrow Checksum

Types:
- Z = zstream()
- Binary = binary()
- Checksum = integer()

Calculate the Adler-32 checksum for Binary.

adler32(Z, PrevAdler, Binary) \rightarrow Checksum

Types:
- Z = zstream()
- PrevAdler = integer()
- Binary = binary()
- Checksum = integer()

Update a running Adler-32 checksum for Binary. If Binary is the empty binary, this function returns the required initial value for the checksum.
Crc = lists:foldl(fun(Bin,Crc0) ->
    zlib:adler32(Z, Crc0, Bin),
    end, zlib:adler32(Z,<<>>, Bins),
compress(Binary) -> Compressed
Types:
    • Binary = Compressed = binary()
Compress a binary (with zlib headers and checksum).
uncompress(Binary) -> Decompressed
Types:
    • Binary = Decompressed = binary()
Uncompress a binary (with zlib headers and checksum).
zip(Binary) -> Compressed
Types:
    • Binary = Compressed = binary()
Compress a binary (without zlib headers and checksum).
unzip(Binary) -> Decompressed
Types:
    • Binary = Decompressed = binary()
Uncompress a binary (without zlib headers and checksum).
gzip(Data) -> Compressed
Types:
    • Binary = Compressed = binary()
Compress a binary (with gz headers and checksum).
gunzip(Bin) -> Decompressed
Types:
    • Binary = Decompressed = binary()
Uncompress a binary (with gz headers and checksum).
The application resource file specifies the resources an application uses, and how the application is started. There must always be one application resource file called Application.app for each application Application in the system. The file is read by the application controller when an application is loaded/stated. It is also used by the functions in systools, for example when generating start scripts.

**FILE SYNTAX**

The application resource file should be called Application.app where Application is the name of the application. The file should be located in the ebin directory for the application.

It must contain one single Erlang term, which is called an application specification:

```
{application, Application, [
  {description, Description},
  {id, Id},
  {vsn, Vsn},
  {modules, Modules},
  {maxP, MaxP},
  {maxT, MaxT},
  {registered, Names},
  {included_applications, Apps},
  {applications, Apps},
  {env, Env},
  {mod, Start},
  {start_phases, Phases}]
}.```

<table>
<thead>
<tr>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>atom()</td>
</tr>
<tr>
<td>Description</td>
<td>string()</td>
</tr>
<tr>
<td>Id</td>
<td>string()</td>
</tr>
<tr>
<td>Vsn</td>
<td>string()</td>
</tr>
<tr>
<td>Modules</td>
<td>[Module]</td>
</tr>
<tr>
<td>MaxP</td>
<td>int()</td>
</tr>
<tr>
<td>MaxT</td>
<td>int()</td>
</tr>
<tr>
<td>Names</td>
<td>[Name]</td>
</tr>
<tr>
<td>Apps</td>
<td>[App]</td>
</tr>
<tr>
<td>Env</td>
<td>[{Par, Val}]</td>
</tr>
<tr>
<td>Start</td>
<td>{Module, StartArgs}</td>
</tr>
<tr>
<td>Phases</td>
<td>[{Phase, PhaseArgs}]</td>
</tr>
</tbody>
</table>

242 Kernel Application (KERNEL)
Module = Name = App = Par = Phase = atom()
Val = StartArgs = PhaseArgs = term()

Application is the name of the application.
For the application controller, all keys are optional. The respective default values are
used for any omitted keys.
The functions in systools require more information. If they are used, the following
keys are mandatory: description, vsn, modules, registered and applications. The
other keys are ignored by systools.

description A one-line description of the application.
id Product identification, or similar.
vsn The version of the application.
modules All modules introduced by this application. systools uses this list when
generating start scripts and tar files. A module can only be defined in one
application.
maxP Deprecated - will be ignored
The maximum number of processes allowed in the application.
maxT The maximum time in milliseconds that the application is allowed to run. After
the specified time the application will automatically terminate.
registered All names of registered processes started in this application. systools uses
this list to detect name clashes between different applications.
included applications All applications which are included by this application. When
this application is started, all included application will automatically be loaded, but
not started, by the application controller. It is assumed that the topmost supervisor
of the included application is started by a supervisor of this application.
applications All applications which must be started before this application is allowed
to be started. systools uses this list to generate correct start scripts. Defaults to
the empty list, but note that all applications have dependencies to (at least)
kernel and stdlib.
env Configuration parameters used by the application. The value of a configuration
parameter is retrieved by calling application:get_env/1,2. The values in the
application resource file can be overridden by values in a configuration file (see
config(4)) or by command line flags (see erl(1)).
mod Specifies the application callback module and a start argument, see
application(3).
The mod key is necessary for an application implemented as a supervision tree, or
the application controller will not know how to start it. The mod key can be
omitted for applications without processes, typically code libraries such as the
application STD LIB.
start_phases A list of start phases and corresponding start arguments for the
application. If this key is present, the application master will - in addition to the
usual call to Module:start/2 - also call
Module:start_phase(Phase,Type,PhaseArgs) for each start phase defined by
the start_phases key, and only after this extended start procedure will
application:start(Application) return.
Start phases may be used to synchronize startup of an application and its included
applications. In this case, the mod key must be specified as:
{mod, {application_starter,[Module,StartArgs]}}
The application master will then call `Module:start/2` for the primary application, followed by calls to `Module:start_phase/3` for each start phase (as defined for the primary application) both for the primary application and for each of its included application, for which the start phase is defined.

This implies that for an included application, the set of start phases must be a subset of the set of phases defined for the primary application. Refer to OTP Design Principles for more information.

SEE ALSO

application(3) [page 34], systools(3)
A configuration file contains values for configuration parameters for the applications in the system. The `erl` command line argument `-config Name` tells the system to use data in the system configuration file `Name.config`.

Configuration parameter values in the configuration file will override the values in the application resource files (see `app(4)`). The values in the configuration file can be overridden by command line flags (see `erl(1)`).

The value of a configuration parameter is retrieved by calling `application:get_env/1,2`.

**FILE SYNTAX**

The configuration file should be called `Name.config` where `Name` is an arbitrary name. The `.config` file contains one single Erlang term. The file has the following syntax:

```
[{Application1, [{Par11, Val11}, ..}], ..
 {ApplicationN, [{ParN1, ValN1}, ..]}].
```

- `Application = atom()` is the name of the application.
- `Par = atom()` is the name of a configuration parameter.
- `Val = term()` is the value of a configuration parameter.

**sys.config**

When starting Erlang in embedded mode, it is assumed that exactly one system configuration file is used, named `sys.config`. This file should be located in `$ROOT/releases/Vsn`, where `$ROOT` is the Erlang/OTP root installation directory and `Vsn` is the release version.

Release handling relies on this assumption. When installing a new release version, the new `sys.config` is read and used to update the application configurations.

This means that specifying another, or additional, `.config` files would lead to inconsistent update of application configurations. Therefore, in Erlang 5.4/OTP R10B, the syntax of `sys.config` was extended to allow pointing out other `.config` files:

```
[{Application, [{Par, Val}]} | File].
```

- `File = string()` is the name of another `.config` file. The extension `.config` may be omitted. It is recommended to use absolute paths. A relative path is relative the current working directory of the emulator.
When traversing the contents of `sys.config` and a filename is encountered, its contents are read and merged with the result so far. When an application configuration tuple `{Application, Env}` is found, it is merged with the result so far. Merging means that new parameters are added and existing parameter values overwritten. Example:

```erlang
sys.config:

[{myapp, [{par1, val1}, {par2, val2}]}],
"/home/user/myconfig"].

myconfig.config:

[{myapp, [{par2, val3}, {par3, val4}]}].
```

This will yield the following environment for `myapp`:

```erlang
[{par1, val1}, {par2, val3}, {par3, val4}]
```

The behaviour if a file specified in `sys.config` does not exist or is erroneous in some other way, is backwards compatible. Starting the runtime system will fail. Installing a new release version will not fail, but an error message is given and the erroneous file is ignored.

SEE ALSO

`app(4)`, `erl(1)`, `OTP Design Principles`
Index of Modules and Functions

Modules are typed in this way. Functions are typed in this way.

abcast/2
   rpc, 220
abcast/3
   rpc, 220
abs/1
   erlang, 74
accept/1
   gen_tcp, 170
accept/2
   gen_tcp, 170
accessible_logs/0
   disk_log, 55
add_path/1
   code, 47
add_patha/1
   code, 47
add_paths/1
   code, 47
add_pathsa/1
   code, 47
add_pathsz/1
   code, 47
add_pathz/1
   code, 47
add_report_handler/1
   error_logger, 144
add_report_handler/2
   error_logger, 144
add_slave/1
   erl_boot_server, 67
adler32/2
   zlib, 240
adler32/3
   zlib, 240
all_loaded/0
   code, 50
allow/1
   net_kernel, 204
alog/2
   disk_log, 55
alog_terms/2
   disk_log, 55
application
   get_all_env/0, 34
   get_all_env/1, 34
   get_all_key/0, 34
   get_all_key/1, 34
   get_application/0, 35
   get_application/1, 35
   get_env/1, 35
   get_env/2, 35
   get_key/1, 35
   get_key/2, 35
   load/1, 35
   load/2, 35
   loaded_applications/0, 36
   Module:config_change/3, 42
   Module:prep_stop/1, 41
   Module:start/2, 40
   Module:start_phase/3, 41
   Module:stop/1, 42
   permit/2, 36
   set_env/3, 37
   set_env/4, 37
   start/1, 37
   start/2, 37
   start_type/0, 38
   stop/1, 38
   takeover/2, 39
   unload/1, 39
unset_env/3, 39
which_applications/0, 40
which_applications/1, 40
apply/2
   erlang, 75
apply/3
   erlang, 75
async_call/4
   rpc, 217
atom_to_list/1
   erlang, 76
auth
   cookie/0, 43
   cookie/1, 43
   is_auth/1, 43
   node_cookie/2, 43
balog/2
   disk_log, 55
balog_terms/2
   disk_log, 55
bchunk/2
   disk_log, 57
bchunk/3
   disk_log, 57
binary_to_list/1
   erlang, 76
binary_to_list/3
   erlang, 76
binary_to_term/1
   erlang, 76
block/1
   disk_log, 56
block/2
   disk_log, 56
block_call/4
   rpc, 216
block_call/5
   rpc, 217
blog/2
   disk_log, 61
blog_terms/2
   disk_log, 61
boot/1
init, 196
breopen/3
   disk_log, 65
btruncate/2
   disk_log, 66
call/4
   rpc, 216
call/5
   rpc, 216
cast/4
   rpc, 219
change_group/2
   file, 146
change_header/2
   disk_log, 56
change_notify/3
   disk_log, 56
change_owner/2
   file, 146
change_owner/3
   file, 147
change_size/2
   disk_log, 56
change_time/2
   file, 147
change_time/3
   file, 147
check_process_code/2
   erlang, 77
chunk/1
   wrap_log_reader, 232
chunk/2
   disk_log, 57
   wrap_log_reader, 232
chunk/3
   disk_log, 57
chunk_info/1
   disk_log, 58
chunk_step/3
   disk_log, 58
clash/0
   code, 52
clear_cmd/0
  heart, 186

close/1
disk_log , 59
  file , 147
gen_tcp , 171
gen_udp , 175
inet , 188
wrap_log_reader , 233
zlib , 235
cmd/1
  os, 208
code
  add_path/1, 47
  add_patha/1, 47
  add_paths/1, 47
  add_pathsa/1, 47
  add_pathsz/1, 47
  add_pathz/1, 47
  all_loaded/0, 50
  clash/0, 52
  compiler_dir/0, 51
  del_path/1, 48
  delete/1, 49
  ensure_loaded/1, 49
  get_object_code/1, 50
  get_path/0, 47
  is_loaded/1, 50
  lib_dir/0, 51
  lib_dir/1, 51
  load_abs/1, 48
  load_binary/3, 49
  load_file/1, 48
  objfile_extension/0, 52
  priv_dir/1, 51
  purge/1, 49
  rehash/0, 52
  replace_path/2, 48
  root_dir/0, 51
  set_path/1, 46
  soft_purge/1, 50
  stick_dir/1, 52
  unstick_dir/1, 52
  where_is_file/1, 52
  which/1, 50
compiler_dir/0
code, 51
compress/1
  zlib, 241
concat_binary/1
erlang, 77
connect/3
gen_tcp , 169
connect/4
gen_tcp , 169
connect_node/1
  net_kernel , 204
consult/1
  file, 147
controlling_process/2
gen_tcp , 171
gen_udp , 174
cookie/0
  auth, 43
cookie/1
  auth, 43
copy/2
  file, 148
copy/3
  file, 148
crc32/1
  zlib, 240
crc32/2
  zlib, 240
crc32/3
  zlib, 240
crc32/4
  zlib, 240
create/1
  pg2, 214
date/0
  erlang, 77
deflate/2
  zlib, 236
deflate/3
  zlib, 236
deflateEnd/1
  zlib, 238
deflateInit/1
  zlib, 235
deflateInit/2
  zlib, 235
deflateInit/6
  zlib, 235
deflateParams/3  zlib, 237
deflateReset/1  zlib, 237
deflateSetDictionary/2  zlib, 237
del_dir/1  file, 148
del_lock/1  global, 177
del_lock/2  global, 177
del_path/1  code, 48
delete/1  code, 49  file, 149  pg2, 214
delete_module/1  erlang, 77
delete_report_handler/1  error_logger, 144
delete_slave/1  erl_boot_server, 67
disconnect_node/1  erlang, 79
disk_log
  accessible_logs/0, 55
  alog/2, 55
  alog_terms/2, 55
  balog/2, 55
  balog_terms/2, 55
  bchunk/2, 57
  bchunk/3, 57
  block/1, 56
  block/2, 56
  blog/2, 61
  blog_terms/2, 61
  breopen/3, 65
  btruncate/2, 66
  change_header/2, 56
  change_notify/3, 56
  change_size/2, 56
  chunk/2, 57
  chunk/3, 57
  chunk_info/1, 58
  chunk_step/3, 58
close/1, 59
format_error/1, 59
inc_wrap_file/1, 59
info/1, 59
lclose/1, 61
lclose/2, 61
log/2, 61
log_terms/2, 61
open/1, 62
pid2name/1, 65
reopen/2, 65
reopen/3, 65
sync/1, 65
truncate/1, 66
truncate/2, 66
unblock/1, 66
dns_hostname/1
  net_adm, 201

element/2
  erlang, 79
ensure_loaded/1
  code, 49
erase/0
  erlang, 80
erase/1
  erlang, 80
erl_boot_server
  add_slave/1, 67
  delete_slave/1, 67
  start/1, 67
  start_link/1, 67
  which_slaves/0, 68
erl_ddll
  format_error/1, 69
  load_driver/2, 69
  loaded_drivers/0, 69
  unload_driver/1, 69
erl_prim_loader
  get_file/1, 71
  get_path/0, 72
  set_path/1, 72
  start/3, 71
erlang
  abs/1, 74
  apply/2, 75
  apply/3, 75
  atom_to_list/1, 76
  binary_to_list/1, 76
is_port/1,90
is_process_alive/1,90
is_record/2,91
is_record/3,91
is_reference/1,91
is_tuple/1,91
length/1,92
link/1,92
list_to_atom/1,92
list_to_binary/1,92
list_to_existing_atom/1,93
list_to_float/1,93
list_to_tuple/1,93
length/1,92
link/1,92
list_to_atom/1,92
list_to_binary/1,92
list_to_existing_atom/1,93
list_to_float/1,93
list_to_tuple/1,93
load_module/2,94
make_ref/0,96
module_loaded/1,98
monitor_node/2,100
node/0,100
node/1,101
nodes/0,101
nodes/1,101
now/0,101
open_port/2,102
pid_to_list/1,104
port_close/1,105
port_command/2,105
port_connect/2,105
port_control/3,106
pre_loaded/0,108
process_flag/2,108
process_flag/3,109
process_info/1,109
process_info/2,110
processes/0,111
purge_module/1,111
put/2,111
register/2,113
registered/0,113
round/1,113
self/0,114
setelement/3,117
size/1,117
spawn/1,117
spawn/2,117
spawn/3,117
spawn/4,118
spawn_link/1,118
spawn_link/2,118
spawn_link/3,118
spawn_link/4,119
spawn_opt/2,119
spawn_opt/3,119
spawn_opt/4,119
spawn_opt/5,120
split_binary/2,121
statistics/1,122
term_to_binary/1,128
term_to_binary/2,128
throw/1,128
time/0,128
tl/1,129
trunc/1,135
tuple_to_list/1,135
unlink/1,136
unregister/1,137
whereis/1,137
erlang:append_element/2
erlang,74
erlang:bump_reductions/1
erlang,76
erlang:cancel_timer/1
erlang,76
erlang:demonitor/1
erlang,78
erlang:demonitor/2
erlang,78
erlang:display/1
erlang,79
erlang:error/1
erlang,80
erlang:error/2
erlang,80
erlang:fault/1
erlang,81
erlang:fault/2
erlang,81
erlang:fun_info/1
erlang,82
erlang:fun_info/2
erlang,83
erlang:fun_to_list/1
erlang,83
erlang:function_exported/3
erlang,84
erlang:get_cookie/0
erlang,85
erlang:get_stacktrace/0
  erlang, 85
erlang:hash/2
  erlang, 86
erlang:hibernate/3
  erlang, 87
erlang:info/1
  erlang, 87
erlang:integer_to_list/2
  erlang, 88
erlang:is_builtin/3
  erlang, 89
erlang:list_to_integer/2
  erlang, 93
erlang:loaded/0
  erlang, 95
erlang:localtime/0
  erlang, 95
erlang:localtime_to_universaltime/2
  erlang, 95
erlang:localtime_to_universaltime/3
  erlang, 95
erlang:make_tuple/2
  erlang, 96
erlang:md5/1
  erlang, 96
erlang:md5_final/1
  erlang, 96
erlang:md5_init/0
  erlang, 97
erlang:md5_update/2
  erlang, 97
erlang:memory/0
  erlang, 97
erlang:memory/1
  erlang, 98
erlang:monitor/2
  erlang, 99
erlang:phash/2
  erlang, 104
erlang:phash2/2
  erlang, 104
erlang:port_call/3
  erlang, 106
erlang:port_info/1
  erlang, 107
erlang:port_info/2
  erlang, 107
erlang:port_to_list/1
  erlang, 107
erlang:ports/0
  erlang, 108
erlang:process_display/2
  erlang, 108
erlang:raise/3
  erlang, 111
erlang:read_timer/1
  erlang, 112
erlang:ref_to_list/1
  erlang, 112
erlang:resume_process/1
  erlang, 113
erlang:send/2
  erlang, 114
erlang:send/3
  erlang, 114
erlang:send_after/3
  erlang, 115
erlang:send_nosuspend/2
  erlang, 115
erlang:send_nosuspend/3
  erlang, 116
erlang:set_cookie/2
  erlang, 116
erlang:start_timer/3
  erlang, 121
erlang:suspend_process/1
  erlang, 122
erlang:system_flag/2
  erlang, 123
erlang:system_info/1
  erlang, 123
erlang:system_monitor/0
  erang, 126
erlang:system_monitor/2
  erlang, 127
erlang:trace/3
  erlang, 129
erlang:trace_info/2
  erlang, 132
erlang:trace_pattern/2
  erlang, 133
erlang:trace_pattern/3
  erlang, 133
erlang:universaltime/0
  erlang, 135
erlang:universaltime_to_localtime/2
  erlang, 136
erlang:yield/0
  erlang, 137
error_handler
  undefined_function/3, 138
    undefined_lambda/3, 138
error_logger
  add_report_handler/1, 144
  add_report_handler/2, 144
  delete_report_handler/1, 144
  error_msg/1, 140
  error_msg/2, 140
  error_report/1, 141
  error_report/2, 141
  format/2, 140
  info_msg/1, 143
  info_msg/2, 143
  info_report/1, 143
  info_report/2, 143
  logfile/1, 144
  tty/1, 144
  warning_map/0, 141
  warning_msg/1, 142
  warning_msg/2, 142
  warning_report/1, 142
  warning_report/2, 142
error_msg/1
  error_logger, 140
error_msg/2
  error_logger, 140
error_report/1
  error_logger, 141
error_report/2
  error_logger, 141
eval/1
  file, 149
eval/2
  file, 149
eval_everywhere/3
  rpc, 219
eval_everywhere/4
  rpc, 219
exit/1
  erlang, 81
exit/2
  erlang, 81
file
  change_group/2, 146
  change_owner/2, 146
  change_owner/3, 147
  change_time/2, 147
  change_time/3, 147
  close/1, 147
  consult/1, 147
  copy/2, 148
  copy/3, 148
  del_dir/1, 148
  delete/1, 149
  eval/1, 149
  eval/2, 149
  file_info/1, 150
  format_error/1, 150
  get_cwd/0, 150
  get_cwd/1, 150
  list_dir/1, 151
  make_dir/1, 151
  make_link/2, 151
  make_symlink/2, 152
  open/2, 152
  path_consult/2, 154
  path_eval/2, 154
  path_open/3, 155
  path_script/2, 155
  path_script/3, 156
  pid2name/1, 156
  position/2, 156
  pread/2, 157
  pread/3, 157
  pwrite/2, 157
  pwrite/3, 158
  read/2, 158
  read_file/1, 158
  read_file_info/1, 159
  read_link/1, 160
  read_link_info/1, 160
  rename/2, 160
script/1, 161
script/2, 161
set_cwd/1, 162
sync/1, 162
truncate/1, 162
write/2, 162
write_file/2, 163
write_file/3, 163
write_file_info/2, 163

cwd/1
file_info/1

find_executable/1
os, 208

find_executable/2
os, 208

float/1
erlang, 82
float_to_list/1
erlang, 82

format/2
error_logger, 140

format_error/1
disk_log, 59
erlDll, 69
file, 150
inet, 189

garbage_collect/0
erlang, 84
garbage_collect/1
erlang, 84
gen_tcp
accept/1, 170
accept/2, 170
close/1, 171
connect/3, 169
connect/4, 169
controlling_process/2, 171
listen/2, 170
recv/2, 171
recv/3, 171
send/2, 171
shutdown/2, 172

gen_udp
close/1, 175
controlling_process/2, 174
open/1, 173
open/2, 173

recv/2, 174
recv/3, 174
send/4, 174

get/0
erlang, 84
get/1
erlang, 84

get_all_env/0
application, 34
get_all_env/1
application, 34

get_all_key/0
application, 34
get_all_key/1
application, 34

get_all_key/2
application, 34

get_application/0
application, 35
get_application/1
application, 35

get_arg/0
init, 196
get_argument/1
init, 196

get_args/0
init, 197
get_arguments/0
init, 197

garbage_collect/0

garbage_collect/1

get_closest_pid/1
pg2, 214

get_cmd/0
heart, 186

getcwd/0
file, 150

getcwd/1
file, 150

get_env/1
application, 35

get_env/2
application, 35

get_file/1
erl_prim_loader, 71

get_key/1
application, 35

get_key/2
application, 35

Kernel Application (KERNEL)
get_keys/1  
erlang, 85
get_local_members/1  
pg2, 215
get_members/1  
pg2, 215
get_net_ticktime/0  
net_kernel, 206
get_object_code/1  
code, 50
get_path/0  
code, 47  
erl_prim_loader, 72
get_plain_arguments/0  
init, 197
get_rc/0  
inet, 188
get_status/0  
init, 197
get_system_tracer/0  
seq_trace, 225
get_token/0  
seq_trace, 224
get_token/1  
seq_trace, 224
getaddr/2  
inet, 189
getaddrs/2  
inet, 189
getBufSize/1  
zlib, 239
getenv/0  
os, 208
getenv/1  
os, 209
gethostbyaddr/1  
inet, 189
gethostbyname/1  
inet, 189
gethostbyname/2  
inet, 189
gethostname/0  
inet, 189
getopts/2  
inet, 190
getpid/0  
os, 209

global

del_lock/1, 177
del_lock/2, 177
notify_all_name/3, 177
random_exit_name/3, 177
random_notify_name/3, 177
re_register_name/2, 178
re_register_name/3, 178
register_name/2, 178
register_name/3, 178
registered_names/0, 178
send/2, 179
set_lock/1, 179
set_lock/2, 179
set_lock/3, 179
sync/0, 180
trans/2, 180
trans/3, 180
trans/4, 180
unregister_name/1, 180
whereis_name/1, 180

global_group

global_groups/0, 182
info/0, 182
monitor_nodes/1, 182
own_nodes/0, 182
registered_names/1, 183
send/2, 183
send/3, 183
sync/0, 183
whereis_name/1, 183
whereis_name/2, 183

global_groups/0  
global_group, 182

group_leader/0  
erlang, 85

group_leader/2  
erlang, 86

gunzip/1  
zlib, 241
gzip/1  
zlib, 241

halt/0  
erlang, 86
halt/1
  erlang, 86
hd/1
  erlang, 86
heart
clear_cmd/0, 186
get_cmd/0, 186
set_cmd/1, 186
host_file/0
  netadm, 201
inc_wrap_file/1
disk_log, 59
inet
close/1, 188
format_error/1, 189
get_rc/0, 188
getaddr/2, 189
getaddr/2, 189
gethostbyaddr/1, 189
gethostbyname/1, 189
gethostbyname/2, 189
gethostname/0, 189
getopts/2, 190
peernamex/1, 190
port/1, 190
setopts/2, 190
sockname/1, 190
inflate/2
  zlib, 238
inflateEnd/1
  zlib, 239
inflateInit/1
  zlib, 238
inflateInit/2
  zlib, 238
inflateReset/1
  zlib, 239
inflateSetDictionary/2
  zlib, 238
info/0
global_group, 182
info/1
disk_log, 59
info_msg/1
  error_logger, 143
info_msg/2
  error_logger, 143
info_report/1
  error_logger, 143
info_report/2
  error_logger, 143
init
  boot/1, 196
get_args/0, 196
get_argument/1, 196
get_arguments/0, 197
get_plain_arguments/0, 197
get_status/0, 197
reboot/0, 198
restart/0, 198
script_id/0, 198
stop/0, 198
integer_to_list/1
  erlang, 87
iolist_size/1
  erlang, 88
iolist_to_binary/1
  erlang, 88
is_alive/0
  erlang, 88
is_atom/1
  erlang, 88
is_auth/1
  auth, 43
is_binary/1
  erlang, 88
is_boolean/1
  erlang, 89
is_float/1
  erlang, 89
is_function/1
  erlang, 89
is_function/2
  erlang, 89
is_integer/1
  erlang, 90
is_list/1
  erlang, 90
is_loaded/1
code, 50
is_number/1
  erlang, 90

is_pid/1
  erlang, 90

is_port/1
  erlang, 90

is_process_alive/1
  erlang, 90

is_record/2
  erlang, 91

is_record/3
  erlang, 91

is_reference/1
  erlang, 91

is_tuple/1
  erlang, 91

join/2
  pg2, 215

lclose/1
  disk_log, 61

lclose/2
  disk_log, 61

leave/2
  pg2, 215

length/1
  erlang, 92

lib_dir/0
  code, 51

lib_dir/1
  code, 51

link/1
  erlang, 92

list_dir/1
  file, 151

list_to_atom/1
  erlang, 92

list_to_binary/1
  erlang, 92

list_to_existing_atom/1
  erlang, 93

list_to_float/1
  erlang, 93

list_to_integer/1
  erlang, 93

list_to_pid/1
  erlang, 94

list_to_tuple/1
  erlang, 94

listen/2
  gen_tcp, 170

load/1
  application, 35

load/2
  application, 35

load_abs/1
  code, 48

load_binary/3
  code, 49

load_driver/2
  erl_dll, 69

load_file/1
  code, 48

load_module/2
  erlang, 94

loaded_applications/0
  application, 36

loaded_drivers/0
  erl_dll, 69

localhost/0
  netadm, 201

log/2
  disk_log, 61

log_terms/2
  disk_log, 61

logfile/1
  error_logger, 144

make_dir/1
  file, 151

make_link/2
  file, 151

make_ref/0
  erlang, 96

make_symmlink/2
  file, 152

258  Kernel Application (KERNEL)
erlang, 102

os
  cmd/1, 208
  find_executable/1, 208
  find_executable/2, 208
  getenv/0, 208
  getenv/1, 209
  getpid/0, 209
  putenv/2, 209
  type/0, 209
  version/0, 209

own_nodes/0
  global_group, 182

packages
  no functions exported, 213

parallel_eval/1
  rpc, 222

path_consult/2
  file, 154

path_eval/2
  file, 154

path_open/3
  file, 155

path_script/2
  file, 155

path_script/3
  file, 156

peername/1
  inet, 190

permit/2
  application, 36

pg2
  create/1, 214
  delete/1, 214
  get_closest_pid/1, 214
  get_local_members/1, 215
  get_members/1, 215
  join/2, 215
  leave/2, 215
  start/0, 215
  start_link/0, 215
  which_groups/0, 215

pid2name/1
  disk_log, 65
  file, 156

pid_to_list/1

erlang, 104

pinfo/1
  rpc, 222

pinfo/2
  rpc, 222

ping/1
  net_adm, 202

pmap/4
  rpc, 222

port/1
  inet, 190

port_close/1
  erlang, 105

port_command/2
  erlang, 105

port_connect/2
  erlang, 105

port_control/3
  erlang, 106

position/2
  file, 156

pre_loaded/0
  erlang, 108

pread/2
  file, 157

pread/3
  file, 157

print/1
  seq_trace, 224

print/2
  seq_trace, 224

priv_dir/1
  code, 51

process_flag/2
  erlang, 108

process_flag/3
  erlang, 109

process_info/1
  erlang, 109

process_info/2
  erlang, 110

processes/0
  erlang, 111

260 Kernel Application (KERNEL)
purge/1
code, 49

purge_module/1
erlang, 111

put/2
erlang, 111

putenv/2
os, 209

put/3
file, 157

random_exit_name/3
global, 177

random_notify_name/3
global, 177

re_register_name/2
global, 178

re_register_name/3
global, 178

read/2
file, 158

read_file/1
file, 158

read_file_info/1
file, 159

read_link/1
file, 160

read_link_info/1
file, 160

reboot/0
init, 198

recv/2
gen_tcp, 171
gen_udp, 174

recv/3
gen_tcp, 171
gen_udp, 174

register/2
erlang, 113

register_name/2
global, 178

register_name/3
safe_multi_server_call/2, 221
safe_multi_server_call/3, 221
sbcast/2, 220
sbcast/3, 220
server_call/4, 220
yield/1, 217

safe_multi_server_call/2
   rpc, 221
safe_multi_server_call/3
   rpc, 221
sbcast/2
   rpc, 220
sbcast/3
   rpc, 220
script/1
   file, 161
script/2
   file, 161
script_id/0
   init, 198
self/0
   erlang, 114
send/2
   gen_tcp, 171
global, 179
global_group, 183
send/3
   global_group, 183
send/4
   gen_udp, 174
seq_trace
   get_system_tracer/0, 225
   get_token/0, 224
   get_token/1, 224
   print/1, 224
   print/2, 224
   reset_trace/0, 225
   set_system_tracer/1, 225
   set_token/1, 223
   set_token/2, 223
server_call/4
   rpc, 220
set_cmd/1
   heart, 186
set_cwd/1
   file, 162
set_env/3
   application, 37
set_env/4
   application, 37
set_lock/1
   global, 179
set_lock/2
   global, 179
set_lock/3
   global, 179
set_net_ticktime/1
   net_kernel, 206
set_net_ticktime/2
   net_kernel, 206
set_path/1
   code, 46
erl_prim_loader, 72
set_system_tracer/1
   seq_trace, 225
set_token/1
   seq_trace, 223
set_token/2
   seq_trace, 223
setBufSize/2
   zlib, 239
setelement/3
   erlang, 117
setopts/2
   inet, 190
shutdown/2
   gen_tcp, 172
size/1
   erlang, 117
sockname/1
   inet, 190
soft_purge/1
   code, 50
spawn/1
   erlang, 117
spawn/2
   erlang, 117
spawn/3
disk_log, 66
uncompress/1
zlib, 241
undefined_function/3
error_handler, 138
undefined lambda/3
error_handler, 138
unlink/1
erlang, 136
unload/1
application, 39
unload_driver/1
erلد, 69
unregister/1
erlang, 137
unregister_name/1
global, 180
unset_env/2
application, 39
unset_env/3
application, 39
unstick_dir/1
code, 52
unzip/1
zlib, 241
version/0
os, 209
warning_map/0
error_logger, 141
warning_msg/1
error_logger, 142
warning_msg/2
error_logger, 142
warning_report/1
error_logger, 142
warning_report/2
error_logger, 142
where is file/1
code, 52
whereis/1
erlang, 137
whereis_name/1
global, 180
global group, 183
whereis_name/2
global group, 183
which/1
code, 50
which applications/0
application, 40
which applications/1
application, 40
which groups/0
pg2, 215
which slaves/0
erld boot server, 68
world/0
netadm, 202
world/1
netadm, 202
world_list/1
netadm, 202
world_list/2
netadm, 202
wrap_log_reader
chunk/1, 232
chunk/2, 232
close/1, 233
open/1, 233
open/2, 233
write/2
file, 162
write file/2
file, 163
write file/3
file, 163
write file info/2
file, 163
yield/1
rpc, 217
zip/1
zlib, 241
zlib
adler32/2, 240
adler32/3, 240
close/1, 235
crc32/1, 240
crc32/2, 240
crc32/3, 240
deflate/2, 236
deflate/3, 236
deflateEnd/1, 238
deflateInit/1, 235
deflateInit/2, 235
deflateInit/6, 235
deflateParams/3, 237
deflateReset/1, 237
deflateSetDictionary/2, 237
getBufSize/1, 239
gunzip/1, 241
gzip/1, 241
inflate/2, 238
inflateEnd/1, 239
inflateInit/1, 238
inflateInit/2, 238
inflateReset/1, 239
inflateSetDictionary/2, 238
open/0, 235
setBufSize/2, 239
uncompress/1, 241
unzip/1, 241
zip/1, 241