Standard Libraries Application (STDLIB)

version 1.8

OTP Team

1997-05-02

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

STDLIB Reference Manual

Short Summaries

c

The following functions are exported:

- `bt(Pid) -> void()`
  [page 32] Evaluates `erlang:process_display(Pid, backtrace)`
- `c(File) -> CompileResult`
  [page 32] Compiles a file
- `c(File, Flags) -> CompileResult`
  [page 32] Compiles a file
- `cd(Dir) -> void()`
  [page 32] Changes directory
- `flush() -> void()`
  [page 33] Flushes the shell message queue
- `help() -> void()`
  [page 33] Displays help information
- `i() -> void()`
  [page 33] Displays system information
- `zi() -> void()`
  [page 33] Displays system information including zombies
- `ni() -> void()`
  [page 33] Displays network information
- `i(X, Y, Z) -> void()`
  [page 33] Evaluates `process_info(pid(X, Y, Z))`
- `l(Module) -> void()`
  [page 33] Loads code into the system

calendar

The following functions are exported:

- `is_leap_year(Year) -> bool()`
  [page 37] Checks if a year is a leap year.
- `last_day_of_the_month(Year, Month) -> int()`
  [page 37] Computes the number of days in a month
- `local_time() -> {Date, Time}`
  [page 38] Computes local time
- `local_time_to_universal_time({Date, Time}) -> {Date, Time}`
  [page 38] Converts from local time to universal time.
- `now_to_local_time(Now) -> {Date, Time}`
  [page 38] Converts now to local date and time
- `now_to_universal_time(Now) -> {Date, Time}`
  [page 38] Converts now to date and time
- `now_to_datetime(Now) -> {Date, Time}`
  [page 38] Converts now to date and time
- `seconds_to_daystime(Secs) -> {Days, Time}`
  [page 38] Computes a days and time from seconds.
- `seconds_to_time(Secs) -> Time`
  [page 39] Computes time from seconds.
- `time_difference(T1, T2) -> Tdiff`
  [page 39] Computes the difference between two times
- `time_to_seconds(Time) -> Secs`
  [page 39] Computes the number of seconds since midnight up to the given time.
- `universal_time() -> {Date, Time}`
  [page 39] Computes universal time
- `universal_time_to_local_time({Date, Time}) -> {Date, Time}`
  [page 39] Converts from universal time to local time.
- `valid_date(Date) -> bool()`
  [page 40] Checks if a date is valid
- `valid_date(Year, Month, Day) -> bool()`
  [page 40] Checks if a date is valid

dets

The following functions are exported:

- `open_file(Name, Args) -> {ok, Name} | {error, Reason}`
  [page 42] Opens a `dets` file.
- `open_file(Filename) -> ok | {error, Reason}`
  [page 43] Opens an existing `dets` file
- `close(Name) -> ok | {error, Reason}`
  [page 43] Closes a file
- `insert(Name, Object) -> ok | {error, Reason}`
  [page 43] Inserts an `Object` in table `Name`
- `lookup(Name, Key) -> ObjectList | {error, Reason}`
  [page 43] Searches the table `Name` for objects with key `Key`

- `traverse(Name, Fun) -> Return`
  [page 43] Traverses the whole file and applies `Fun`
- `delete(Name, Key) -> ok`
  [page 44] Deletes all objects with a specific key from a table
- `delete_object(Name, Object) -> ok`
  [page 44] Deletes a specific object from a table
- `first(Name) -> Key | '$end_of_table'`
  [page 44] Returns the 'first' object in a table
- `next(Name, Key) -> Key | '$end_of_table'`
  [page 44] Returns the next key in a table
- `slot(Name, I) -> $end_of_table | ObjList`
  [page 44] Returns the list of objects associated with slot `I`
- `all() -> NameList`
  [page 44] Returns a list of all open files on this node.
- `sync(Name) -> ok`
  [page 44] Ensures that all data written to `Name` is written to disk.
- `match_object(Name, Pattern) -> ObjectList`
  [page 44] Matches objects and returns a list of all objects which match `Pattern`
- `match(Name, Pattern) -> BindingsList`
  [page 45] Matches objects and returns a list of variable bindings
- `match_delete(Name, Pattern) -> ok`
  [page 45] Deletes all objects from `Name`
- `info(Name) -> InfoList`
  [page 45] Returns a list of {`Tag, Value`} pairs describing the file.
- `safe_fixtable(Name, true|false)`
  [page 45] Disables rehashing of a table
- `info(Name, Key) -> Value`
  [page 45] Returns one of the possible information fields which are available by means of `info/1`

dict

The following functions are exported:

- `append(Key, Value, Dict1) -> Dict2`
  [page 47] Appends `value` to `key` in dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`
  [page 47] Appends new values to dictionary
- `dict_to_list(Dict) -> List`
  [page 47] Converts a dictionary to a list
- `erase(Key, Dict1) -> Dict2`
  [page 47] Erases a key from a dictionary
- `fetch(Key, Dict) -> Value`
  [page 47] Look-up values in a dictionary
- `fetch_keys(Dict) -> Keys`
  [page 48] Returns all keys in a dictionary

- `find(Key, Dict) -> Result`
  [page 48] Searches for a key in a dictionary
- `is_key(Key, Dict) -> bool()`
  [page 48] Tests if a key is in a dictionary.
- `new() -> dictionary()`
  [page 48] Creates a dictionary
- `store(Key, Value, Dict1) -> Dict2`
  [page 48] Stores a value in a dictionary

digraph

The following functions are exported:

- `new(Opts) -> graph()`
  [page 50] Creates a new empty graph
- `new() -> graph()`
  [page 50] Returns a protected empty graph, where cycles are allowed
- `delete(G) -> true`
  [page 50] Deletes the graph
- `add_vertex(G, Vertex, Data) -> V`
  [page 50] Adds a new vertex
- `add_vertex(G, Vertex) -> V`
  [page 50] Adds a new vertex
- `add_vertex(G) -> V`
  [page 50] Adds a new vertex
- `vertex(G, V) -> {V,Data} | false`
  [page 51] Lookup a vertex
- `vertices(G) -> Vertices`
  [page 51] Lists all vertices in the graph
- `del_vertex(G, V) -> true`
  [page 51] Deletes a vertex
- `del_vertices(G, Vertices) -> true`
  [page 51] Deletes multiple vertices
- `add_edge(G, Edge, V1, V2, Data) -> E | {error, Reason}`
  [page 51] Adds a directed edge
- `add_edge(G, V1, V2, Data) -> E | {error, Reason}`
  [page 51] Adds a directed edge
- `add_edge(G, V1, V2) -> E | {error, Reason}`
  [page 51] Adds a directed edge
- `del_edge(G, E) -> true`
  [page 52] Deletes an edge
- `del_edges(G, Edges) -> true`
  [page 52] Deletes multiple edges
- `edge(G, E) -> {E, V1, V2, Data} | false`
  [page 52] Lookup an edge

- `edges(G) -> Edges`
  [page 52] Lists all edges in the graph
- `out_neighbours(G, V) -> Vertices`
  [page 52] Lists all vertices which have a directed edge from vertex `V`
- `in_neighbours(G, V) -> Vertices`
  [page 53] Lists all vertices which have a directed edge to vertex `V<>`
- `out_edges(G, V) -> Edges`
  [page 53] Lists all edges that ends in vertex `V`
- `in_edges(G, V) -> Edges`
  [page 53] Lists all edges which either start or end in vertex `V`
- `edges(G, V) -> Edges`
  [page 53]
- `out_degree(G, V) -> int()`
  [page 53] Calculates the out degree of a vertex
- `in_degree(G, V) -> int()`
  [page 53] Calculates the in degree of a vertex
- `del_path(G, V1, V2) -> true`
  [page 54] Deletes paths
- `get_path(G, V1, V2) -> Vertices | false`
  [page 54] Finds one path
- `get_cycle(G, V) -> Vertices | false`
  [page 54] Finds one cycle

epp

The following functions are exported:

- `open(FileName, IncludePath) -> {ok,Epp} | {error, ErrorDescriptor}`
  [page 55] Opens a file for preprocessing
- `open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}`
  [page 55] Opens a file for preprocessing
- `close(Epp) -> ok`
  [page 55] Closes the preprocessing of the file associated with `Epp`
- `parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`
  [page 55] Returns the next Erlang form from the opened Erlang source file
- `parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}`
  [page 55] Preprocesses and parses an Erlang source file

erl_eval

The following functions are exported:

- `exprs(Expressions, Bindings) -> {value, Value, NewBindings}`
  [page 57] Evaluates expressions
- `exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 57] Evaluates expressions
- `expr(Expression, Bindings) -> { value, Value, NewBindings }`
  [page 57] Evaluates expression
- `expr(Expression, Bindings, LocalFunctionHandler) -> { value, Value, NewBindings }`
  [page 57] Evaluates expression
- `expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
  [page 57] Evaluates a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 57] Evaluates a list of expressions
- `new_bindings() -> BindingStruct`
  [page 58] Returns a bindings structure
- `bindings(BindingStruct) -> Bindings`
  [page 58] Returns bindings
- `binding(Name, BindingStruct) -> Binding`
  [page 58] Returns bindings
- `add_binding(Name, Value, Bindings) -> BindingStruct`
  [page 58] Adds a binding
- `del_binding(Name, Bindings) -> BindingStruct`
  [page 58] Deletes a binding

erl_id_trans

The following functions are exported:

- `parse_transform(Forms, Options) -> Forms`
  [page 60] Transforms Erlang forms

erl_internal

The following functions are exported:

- `bif(Name, Arity) -> bool()`
  [page 61] Tests for an Erlang BIF
- `guard_bif(Name, Arity) -> bool()`
  [page 61] Tests for an Erlang BIF allowed in guards

- `type_test(Name, Arity) -> bool()`
  [page 61] Tests for a valid type test
- `arith_op(OpName, Arity) -> bool()`
  [page 61] Tests for an arithmetic operator
- `bool_op(OpName, Arity) -> bool()`
  [page 61] Tests for a Boolean operator
- `comp_op(OpName, Arity) -> bool()`
  [page 62] Tests for a comparison operator
- `list_op(OpName, Arity) -> bool()`
  [page 62] Tests for a list operator
- `send_op(OpName, Arity) -> bool()`
  [page 62] Tests for a send operator
- `op_type(OpName, Arity) -> Type`
  [page 62] Returns operator type

erl_lint

The following functions are exported:

- `module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 63] Checks a module for errors
- `module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 63] Checks a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 63] Checks a module for errors
- `is_guard_test(Expr) -> bool()`
  [page 64] Tests for a guard test
- `format_error(ErrorDescriptor) -> string()`
  [page 64] Formats an error descriptor

erl_parse

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`
  [page 66] Parses an Erlang form
- `parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`
  [page 66] Parses Erlang expressions
- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`
  [page 66] Parses an Erlang term
- `format_error(ErrorDescriptor) -> string()`
  [page 67] Formats an error descriptor

- `tokens(AbsTerm) -> Tokens`
  [page 67] Generates a list of tokens for an expression
- `tokens(AbsTerm, MoreTokens) -> Tokens`
  [page 67] Generates a list of tokens for an expression
- `normalise(AbsTerm) -> Data`
  [page 67] Converts abstract form to an Erlang term
- `abstract(Data) -> AbsTerm`
  [page 67] Converts a Erlang term into an abstract form

erl_pp

The following functions are exported:

- `form(Form) -> DeepCharList`
  [page 69] Pretty prints a form
- `form(Form, HookFunction) -> DeepCharList`
  [page 69] Pretty prints a form
- `attribute(Attribute) -> DeepCharList`
  [page 69] Pretty prints an attribute
- `attribute(Attribute, HookFunction) -> DeepCharList`
  [page 69] Pretty prints an attribute
- `function(Function) -> DeepCharList`
  [page 69] Pretty prints a function
- `function(Function, HookFunction) -> DeepCharList`
  [page 69] Pretty prints a function
- `guard(Guard) -> DeepCharList`
  [page 69] Pretty prints a guard
- `guard(Guard, HookFunction) -> DeepCharList`
  [page 70] Pretty prints a guard
- `exprs(Expressions) -> DeepCharList`
  [page 70] Pretty prints `Expressions`
- `exprs(Expressions, HookFunction) -> DeepCharList`
  [page 70] Pretty prints `Expressions`
- `exprs(Expressions, Indent, HookFunction) -> DeepCharList`
  [page 70] Pretty prints `Expressions`
- `expr(Expression) -> DeepCharList`
  [page 70] Pretty prints one `Expression`
- `expr(Expression, HookFunction) -> DeepCharList`
  [page 70] Pretty prints one `Expression`
- `expr(Expression, Indent, HookFunction) -> DeepCharList`
  [page 70] Pretty prints one `Expression`
- `expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList`
  [page 70] Pretty prints one `Expression`

erl_scan

The following functions are exported:

- `string(CharList,StartLine]) -> {ok, Tokens, EndLine} | Error`
  [page 72] Scans a string and returns the Erlang tokens
- `string(CharList) -> {ok, Tokens, EndLine} | Error`
  [page 72] Scans a string and returns the Erlang tokens
- `tokens(Continuation, CharList, StartLine) ->Return`
  [page 72] Re-entrant scanner
- `reserved_word(Atom) -> bool()`
  [page 73] Tests for a reserved word
- `format_error(ErrorDescriptor) -> string()`
  [page 73] Formats an error descriptor

ets

The following functions are exported:

- `new(Name, Type)`
  [page 75]
- `insert(Tab, Object)`
  [page 75]
- `lookup(Tab, Key)`
  [page 76]
- `lookup_element(Tab, Key, Pos)`
  [page 76] Look-up of element
- `delete(Tab, Key) -> true`
  [page 76]
- `delete(Tab)`
  [page 76]
- `update_counter(Tab, Key, Incr)`
  [page 76] Updates a counter object
- `first(Tab)`
  [page 77]
- `next(Tab, Key)`
  [page 77]
- `last(Tab)`
  [page 77]
- `prev(Tab, Key)`
  [page 77]
- `slot(Tab, I)`
  [page 77]
- `fixtable(Tab, true|false)`
  [page 78]

- `safe_fixtable(Tab, true|false)`
  [page 78]
- `all()`
  [page 79] Returns a list of all tables on this node.
- `match(Tab, Pattern)`
  [page 79]
- `match_object(Tab, Pattern)`
  [page 80] Returns all objects in `Tab` matching `Pattern`
- `match_delete(Tab, Pattern)`
  [page 80]
- `rename(Tab,NewName)`
  [page 80]
- `info(Tab)`
  [page 80]
- `info(Tab, Item)`
  [page 81]
- `tab2file(Tab, Filename)`
  [page 81]
- `file2tab(Filename)`
  [page 81]
- `tab2list(Tab)`
  [page 81]
- `i()`
  [page 81]
- `i(Item)`
  [page 81]

filename

The following functions are exported:

- `absname(Filename) -> Absname`
  [page 82] Converts a relative `Filename` to an absolute name
- `absname(Filename, Directory) -> Absname`
  [page 82] Converts the relative `Filename` to an absolute name, based on `Directory`.
- `basename(Filename)`
  [page 83] Returns the part of the `Filename` after the last directory separator
- `basename(Filename,Ext) -> string()`
  [page 83] Returns the last component of `Filename` with `Ext`stripped
- `dirname(Filename) -> string()`
  [page 83] Returns the directory part of a path name
- `extension(Filename) -> string() | []`
  [page 83] Returns the file extension
- `join(Components) -> string()`
  [page 84] Joins a list of file name `Components` with directory separators

- `join(Name1, Name2) -> string()`
  [page 84] Joins two file name components with directory separators.
- `nativename(Path) -> string()`
  [page 84] Returns the native form of a file Path
- `pathtype(Path) -> absolute | relative | volumerelative`
  [page 84] Returns the type of a Path
- `rootname(Filename) -> string()`
  [page 85] Returns all characters in `Filename`, except the extension.
- `rootname(Filename, Ext) -> string()`
  [page 85] Returns all characters in `Filename`, except the extension.
- `split(Filename) -> Components`
  [page 85] Returns a list whose elements are the file name components of `Filename`.
- `find_src(Module) -> {SourceFile, Options}`
  [page 85] Finds the `Filename` and compilation options for a compiled `Module`.
- `find_src(Module, Rules) -> {SourceFile, Options}`
  [page 85] Finds the `Filename` and compilation options for a compiled `Module`.

gen_event

The following functions are exported:

- `start() -> ServerRet`
  [page 88] Starts an event manager
- `start(Name) -> ServerRet`
  [page 88] Starts an event manager
- `start_link() -> ServerRet`
  [page 88] Starts an event manager
- `start_link(Name) -> ServerRet`
  [page 88] Starts an event manager
- `stop(EventMgr) -> ok`
  [page 88] Terminates the event manager
- `notify(EventMgr, Event) -> ok`
  [page 89] Sends an event notification to an event manager
- `sync_notify(EventMgr, Event) -> ok`
  [page 89] Sends an event notification to an event manager
- `add_handler(EventMgr, Handler, Args) -> ok | ErrorRet`
  [page 89] Adds a new event handler
- `add_sup_handler(EventMgr, Handler, Args) -> ok | ErrorRet`
  [page 89] Adds a new supervised event handler
- `delete_handler(EventMgr, Handler, Args) -> DelRet`
  [page 90] Removes an event handler
- `swap_handler(EventMgr, OldHandler, NewHandler) -> SwRet`
  [page 90] Installs a new event handler in place of the old handler
- `swap_sup_handler(EventMgr, OldHandler, NewHandler) -> SwRet`
  [page 91] Installs a new event handler in place of the old handler

- `call(EventMgr, Handler, Query) -> Ret`
  [page 91] Sends a request to a specific handler
- `call(EventMgr, Handler, Query, Timeout) -> Ret`
  [page 91] Sends a request to a specific handler
- `which_handlers(EventMgr) -> [Handler]`
  [page 92] Which event handlers are active in an event manager
- `Module:init(Args) -> InitRes`
  [page 93] Initializes a new event handler
- `Module:handle_event(Event, State) -> EventRet`
  [page 93] Handles an event in a event handler
- `Module:handle_call(Query, State) -> CallRet`
  [page 93] Handles a request dedicated to the event handler
- `Module:handle_info(Info, State) -> EventRet`
  [page 94] Handles miscellaneous events
- `Module:terminate(Arg, State) -> TermRet`
  [page 95] Cleans up before the event handler is removed
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 95] Changes the state of the event handler

gen_fsm

The following functions are exported:

- `start(Module, StartArgs, Options) -> StartRet`
  [page 97] Starts an FSM process
- `start_link(Module, StartArgs, Options) -> StartRet`
  [page 97] Starts an FSM process
- `start(Name, Module, StartArgs, Options) -> StartRet`
  [page 97] Starts an FSM process
- `start_link(Name, Module, StartArgs, Options) -> StartRet`
  [page 97] Starts an FSM process
- `send_event(ProcessRef,Event) -> void()`
  [page 97] Sends an event asynchronously to the FSM process
- `send_all_state_event(ProcessRef,Event) -> void()`
  [page 98] An event, which can be handled in all states, is sent asynchronously to the FSM process
- `sync_send_event(ProcessRef,Event) -> Reply`
  [page 98] Sends an event synchronously to the FSM process
- `sync_send_event(ProcessRef,Event, Timeout) -> Reply`
  [page 98] Sends an event synchronously to the FSM process
- `sync_send_all_state_event(ProcessRef,Event) -> Reply`
  [page 98] An event, which can be handled in all states, is sent synchronously to the FSM process
- `sync_send_all_state_event(ProcessRef,Event,Timeout) -> Reply`
  [page 98] An event, which can be handled in all states, is sent synchronously to the FSM process

- `reply(To, Reply) -> true`
  [page 99] Sends an explicit reply to a caller
- `Module:init(StartArgs) -> Return`
  [page 99] Initializes the FSM process
- `Module:StateName(Event, StateData) -> Return`
  [page 100] Handles asynchronous events in this state
- `Module:StateName(Event, From, StateData) -> Return`
  [page 100] Handles synchronous events in this state
- `Module:handle_event(Event, StateName, StateData) -> Return`
  [page 101] Handles events common to all states
- `Module:handle_sync_event(Event, From, StateName, StateData) -> Return`
  [page 101] Handles events common to all states
- `Module:handle_info(Info, StateName, StateData) -> Return`
  [page 101] Handles other messages received by the process
- `Module:terminate(Reason, StateName, StateData) -> void()`
  [page 102] Terminates the FSM
- `Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NewState, NewStateData}`
  [page 102] Changes the FSM

gen_server

The following functions are exported:

- `start(Module, Args, Options) -> ServerRet`
  [page 105] Starts a gen_server server.
- `start(ServerName, Module, Args, Options) -> ServerRet`
  [page 105] Starts a gen_server server.
- `start_link(Module, Args, Options) -> ServerRet`
  [page 105] Starts a gen_server server.
- `start_link(ServerName, Module, Args, Options) -> ServerRet`
  [page 105] Starts a gen_server server.
- `call(ServerRef, Request) -> Reply`
  [page 105] Makes a request to a server and waits for the reply
- `call(ServerRef, Request, Timeout) -> Reply`
  [page 105] Makes a request to a server and waits for the reply
- `cast(ServerRef, Request) -> ok`
  [page 106] Casts a request to a server. No reply is expected from the server.
- `multi_call(DistRef, Request) -> DistRep`
  [page 106] Makes a request to a server on several nodes
- `multi_call(Nodes, DistRef, Request) -> DistRep`
  [page 106] Makes a request to a server on several nodes
- `abcast(DistRef, Request) -> abcast`
  [page 106] Casts a request to a server which exists on several nodes

- `abcast(Nodes, DistRef, Request) -> abcast`
  [page 106] Casts a request to a server which exists on several nodes
- `reply(To, Reply) -> true`
  [page 107] Sends a explicit reply to a client
- `Module:init(Args) -> {ok, State} | {ok, State, Timeout} | ignore | {stop, StopReason}`
  [page 107] Initializes the server
- `Module:handle_call(Request, From, State) -> CallReply`
  [page 108] Handles a call request
- `Module:handle_cast(Request, State) -> Return`
  [page 108] Handles a cast request
- `Module:handle_info(Info, State) -> Return`
  [page 109] Handles miscellaneous messages
- `Module:terminate(Reason, State) -> ok`
  [page 109] Cleans up the server before termination
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 110] Changes the state of the server

io

The following functions are exported:

- `put_chars([IoDevice,] Chars)`
  [page 112] Writes characters to standard output
- `nl([IoDevice])`
  [page 112] Outputs a newline
- `get_chars([IoDevice,] Prompt, Count)`
  [page 112] Reads characters from standard input
- `get_line([IoDevice,] Prompt)`
  [page 112] Reads a line from standard input
- `write([IoDevice,] Term)`
  [page 112] Writes a term
- `read([IoDevice,] Prompt)`
  [page 112] Reads a term
- `fwrite(Format)`
  [page 113] Writes formatted output
- `format(Format)`
  [page 113] Writes formatted output
- `fwrite([IoDevice,] Format, Arguments)`
  [page 113] Writes formatted output
- `format([IoDevice,] Format, Arguments)`
  [page 113] Writes formatted output
- `fread([IoDevice,] Prompt, Format)`
  [page 116] Reads formatted input
- `scan_erl_exprs(Prompt)`
  [page 117] Reads Erlang tokens

- `scan_erl_exprs([IoDevice,] Prompt, StartLine)`
  [page 117] Reads Erlang tokens
- `scan_erl_form(Prompt)`
  [page 117] Reads Erlang tokens
- `scan_erl_form(IoDevice, Prompt[, StartLine])`
  [page 117] Reads Erlang tokens
- `parse_erl_exprs(Prompt)`
  [page 117] Reads Erlang expressions
- `parse_erl_exprs(IoDevice, Prompt[, StartLine])`
  [page 117] Reads Erlang expressions
- `parse_erl_form(Prompt)`
  [page 118] Reads Erlang form
- `parse_erl_form(IoDevice, Prompt[, StartLine])`
  [page 118] Reads Erlang form

io_lib

The following functions are exported:

- `nl()`
  [page 119] Returns a newline
- `write(Term)`
  [page 119] Writes a term
- `write(Term, Depth)`
  [page 119] Writes a term
- `print(Term)`
  [page 119] Pretty prints a term
- `print(Term, Column, LineLength, Depth)`
  [page 119] Pretty prints a term
- `fwrite(Format, Data)`
  [page 119] Formatted output
- `format(Format, Data)`
  [page 119] Formatted output
- `fread(Format, String)`
  [page 119] Formatted input
- `fread(Continuation, CharList, Format)`
  [page 120] Re-entrant formatted reader
- `write_atom(Atom)`
  [page 120] Returns an atom
- `write_string(String)`
  [page 120] Returns a string
- `write_char(Integer)`
  [page 120] Returns a character
- `indentation(String, StartIndent)`
  [page 120] Indentation after printing string

- `char_list(CharList) -> bool()`
  [page 120] Tests for a list of characters
- `deep_char_list(CharList)`
  [page 121] Tests for a deep list of characters
- `printable_list(CharList)`
  [page 121] Tests for a list of printable characters

lib

The following functions are exported:

- `flush_receive() -> void()`
  [page 122] Flushes messages
- `error_message(Format, Args)`
  [page 122] Prints error message
- `progname() -> atom()`
  [page 122] Returns Erlang starter
- `nonl(List1)`
  [page 122] Removes last newline
- `send(To, Msg)`
  [page 122] Sends a message
- `sendw(To, Msg)`
  [page 122] Sends a message and waits fo an answer

lists

The following functions are exported:

- `append(ListOfLists) -> List1`
  [page 123] Appends a list of lists
- `append(List1, List2) -> List3`
  [page 123] Appends two lists
- `concat(Things) -> string()`
  [page 123] Concatenates a list of atoms
- `delete(Element, List1) -> List2`
  [page 124] Deletes an element in a list
- `duplicate(N, Element) -> List`
  [page 124] Makes N copies of element
- `flatlength(DeepList) -> int()`
  [page 124] Length of flattened deep list
- `flatten(DeepList) -> List`
  [page 124] Flattens a deep list
- `flatten(DeepList, Tail) -> List`
  [page 124] Flattens a deep list

- `subtract(List1, List2) -> List3`
  [page 128] Subtracts the element in one list from another list
- `suffix(List1, List2) -> bool()`
  [page 129] Tests for list suffix
- `sum(List) -> number()`
  [page 129] Returns sum of elements in a list
- `all(Pred, List) -> bool()`
  [page 129] True if all elements in the list satisfy `Pred`
- `any(Pred, List) -> bool()`
  [page 129] True if any of the elements X in the list satisfies `Pred(X)`
- `dropwhile(Pred, List1) -> List2`
  [page 129] Drops elements from List1 while `Pred` is true
- `filter(Pred, List1) -> List2`
  [page 129] Chooses elements which satisfy a predicate
- `flatmap(Function, List1) -> Element`
  [page 129] Maps and flattens in one pass
- `foldl(Function, Acc0, List) -> Acc1`
  [page 130] Folds a function over a list
- `foldr(Function, Acc0, List) -> Acc1`
  [page 130] Folds a function over a list
- `foreach(Function, List) -> void()`
  [page 130] Applies function to each element of a list
- `map(Func, List1) -> List2`
  [page 130] Maps a function over a list
- `mapfoldl(Function, Acc0, List1) -> {List2, Acc}`
  [page 130] Maps and folds in one pass
- `mapfoldr(Function, Acc0, List1) -> {List2, Acc}`
  [page 131] Maps and folds in one pass
- `splitwith(Pred, List) -> {List1, List2}`
  [page 131] Partitions List1 into two lists according to `Pred`
- `takewhile(Pred, List1) -> List2`
  [page 131] Takes elements from List1 while `Pred` is true

log_mf_h

The following functions are exported:

- `init(Dir, MaxBytes, MaxFiles)`
  [page 133] Initiates the event handler
- `init(Dir, MaxBytes, MaxFiles, Pred) -> Args`
  [page 133] Initiates the event handler

math

The following functions are exported:

- `pi() -> float()`
  [page 134] A useful number
- `sin(X)`
  [page 134] Diverse math functions
- `cos(X)`
  [page 134] Diverse math functions
- `tan(X)`
  [page 134] Diverse math functions
- `asin(X)`
  [page 134] Diverse math functions
- `acos(X)`
  [page 134] Diverse math functions
- `atan(X)`
  [page 134] Diverse math functions
- `atan2(X, Y)`
  [page 134] Diverse math functions
- `sinh(X)`
  [page 134] Diverse math functions
- `cosh(X)`
  [page 134] Diverse math functions
- `tanh(X)`
  [page 134] Diverse math functions
- `asinh(X)`
  [page 134] Diverse math functions
- `acosh(X)`
  [page 134] Diverse math functions
- `atanh(X)`
  [page 134] Diverse math functions
- `exp(X)`
  [page 134] Diverse math functions
- `log(X)`
  [page 134] Diverse math functions
- `log10(X)`
  [page 134] Diverse math functions
- `pow(X, Y)`
  [page 134] Diverse math functions
- `sqrt(X)`
  [page 134] Diverse math functions
- `erf(X) -> float()`
  [page 134] Error function.
- `erfc(X) -> float()`
  [page 135] Another error function

ordsets

The following functions are exported:

- `new_set() -> OrdSet`
  [page 136] Returns an empty set
- `is_set(OrdSet) -> bool()`
  [page 136] Tests for an `Ordset`
- `set_to_list(OrdSet) -> List`
  [page 136] Converts an `Ordset` into a list
- `list_to_set(List) -> OrdSet`
  [page 136] Converts a list into an `Ordset`
- `is_element(Element, OrdSet) -> bool()`
  [page 136] Tests for membership of an `Ordset`
- `add_element(Element, OrdSet1) -> OrdSet2`
  [page 137] Adds an element to an `Ordset`
- `del_element(Element, OrdSet1) -> OrdSet2`
  [page 137] Removes an element from an `Ordset`
- `union(OrdSet1, OrdSet2) -> OrdSet3`
  [page 137] Union of two `Ordsets`
- `union(OrdSetList) -> OrdSet`
  [page 137] Union of a list of `Ordsets`
- `intersection(OrdSet1, OrdSet2) -> OrdSet3`
  [page 137] Intersection of two `Ordsets`
- `intersection(OrdSetList) -> OrdSet`
  [page 137] Intersection of a list of `Ordsets`
- `subtract(OrdSet1, OrdSet2) -> OrdSet3`
  [page 137] Difference of two `Ordsets`
- `subset(OrdSet1, OrdSet2) -> bool()`
  [page 137] Tests for subset

pg

The following functions are exported:

- `create(PgName)`
  [page 139]
- `create(PgName, Node)`
  [page 139]
- `join(PgName, Pid)`
  [page 139]
- `send(Pgname, Message)`
  [page 139]
- `esend(PgName, Mess)`
  [page 139]
- `members(PgName)`
  [page 139]

pool

The following functions are exported:

- `start(Name)`
  [page 140]
- `start(Name, Args)`
  [page 140]
- `attach(Node)`
  [page 140]
- `stop()`
  [page 140]
- `get_nodes()`
  [page 141]
- `pspawn(Mod, Fun, Args)`
  [page 141]
- `pspawn_link(Mod, Fun, Args)`
  [page 141]
- `get_node()`
  [page 141]
- `new_node(Host, Name)`
  [page 141]

proc_lib

The following functions are exported:

- `spawn(Module,Func,Args) -> Pid`
  [page 142] Spawns a new process
- `spawn(Node,Module,Func,Args) -> Pid`
  [page 142] Spawns a new process
- `spawn_link(Module,Func,Args) -> Pid`
  [page 142] Spawns a new process and sets a link
- `spawn_link(Node,Module,Func,Args) -> Pid`
  [page 142] Spawns a new process and sets a link
- `start(Module,Func,Args) -> Ret`
  [page 143] Starts a new process synchronously
- `start(Module,Func,Args,Time) -> Ret`
  [page 143] Starts a new process synchronously
- `start_link(Module,Func,Args) -> Ret`
  [page 143] Starts a new process synchronously
- `start_link(Module,Func,Args,Time) -> Ret`
  [page 143] Starts a new process synchronously
- `init_ack(Parent, Ret) -> void()`
  [page 143] Used by a process when it has started

- `init_ack(Ret) -> void()`
  [page 143] Used by a process when it has started
- `format(CrashReport) -> string()`
  [page 144] Formats a crash report
- `initial_call(PidOrPinfo) -> {Module,Function,Args} | false`
  [page 144] Extracts the initial call of a `proc_lib` spawned process
- `translate_initial_call(PidOrPinfo) -> {Module,Function,Arity}`
  [page 144] Extracts and translates the initial call of a `proc_lib` spawned process

queue

The following functions are exported:

- `new() -> Queue`
  [page 146] Creates a new empty FIFO queue
- `in(Item, Q1) -> Q2`
  [page 146] Inserts an item into a queue
- `out(Q) -> Result`
  [page 146] Removes an item from a queue
- `to_list(Q) -> list()`
  [page 146] Converts a queue to a list

random

The following functions are exported:

- `seed() -> ran()`
  [page 147] Seeds random number generation with default values
- `seed(A1, A2, A3) -> ran()`
  [page 147] Seeds random number generator
- `uniform() -> float()`
  [page 147] Returns a random float
- `uniform(N) -> int()`
  [page 147] Returns a random integer

regexp

The following functions are exported:

- `match(String, RegExp) -> MatchRes`
  [page 148] Matches a regular expression
- `first_match(String, RegExp) -> MatchRes`
  [page 148] Matches a regular expression

- `matches(String, RegExp) -> MatchRes`
  [page 148] Matches a regular expression
- `sub(String, RegExp, New) -> SubRes`
  [page 149] Substitutes the first occurrence of a regular expression
- `gsub(String, RegExp, New) -> SubRes`
  [page 149] Substitutes all occurrences of a regular expression
- `split(String, RegExp) -> SplitRes`
  [page 149] Splits a string into fields
- `sh_to_awk(ShRegExp) -> AwkRegExp`
  [page 150] Converts an `sh` regular expression into an `AWK` one
- `parse(RegExp) -> ParseRes`
  [page 150] Parses a regular expression
- `format_error(ErrorDescriptor) -> string()`
  [page 150] Formats an error descriptor

shell

No functions are exported

shell_default

No functions are exported

slave

The following functions are exported:

- `start(Host)`
  [page 160] Starts a slave node at Host
- `start_link(Host)`
  [page 160] Starts a slave node at Host
- `start(Host, Name)`
  [page 160] Starts a slave node at Host called Name@Host
- `start_link(Host, Name)`
  [page 161] Starts a slave node at Host called Name@Host
- `start(Host, Name, Args) -> {ok, Node} | {error, ErrorInfo}`
  [page 161] Starts a slave node at Host called `Name@Host` and passes `Args` to new node
- `start_link(Host, Name, Args)`
  [page 161] Starts a slave node at Host called Name@Host
- `stop(Node)`
  [page 162]

- `pseudo([Master | ServerList])`
  [page 162] Starts a number of pseudo servers
- `pseudo(Master, ServerList)`
  [page 162] Starts a number of pseudo servers
- `relay(Pid)`
  [page 162]

string

The following functions are exported:

- `len(String) -> Length`
  [page 163] The length of a string
- `equal(String1, String2) -> bool()`
  [page 163] Tests string equality
- `concat(String1, String2) -> String3`
  [page 163] Concatenates two strings
- `chr(String, Character) -> Index`
  [page 163] Finds the index of a character
- `rchr(String, Character) -> Index`
  [page 163] Finds the index of a character
- `str(String, SubString) -> Index`
  [page 163] Finds the index of a substring
- `rstr(String, SubString) -> Index`
  [page 163] Finds the index of a substring
- `span(String, Chars) -> Length`
  [page 164] Spans characters at start of string
- `cspan(String, Chars) -> Length`
  [page 164] Spans characters at start of string
- `substr(String, Start) -> SubString`
  [page 164] Extracts a substring
- `substr(String, Start, Length) -> Substring`
  [page 164] Extracts a substring
- `tokens(String, SeperatorList) -> Tokens`
  [page 164] Splits string into tokens
- `chars(Character, Number) -> String`
  [page 164]
- `chars(Character, Number, Tail) -> String`
  [page 164]
- `copies(String, Number) -> Copies`
  [page 165] Copies a string
- `words(String) -> Count`
  [page 165] Counts blank seperated words
- `words(String, Character) -> Count`
  [page 165] Counts blank seperated words

- `sub_word(String, Number) -> Word`
  [page 165] Extracts subword
- `sub_word(String, Number, Character) -> Word`
  [page 165] Extracts subword
- `strip(String) -> Stripped`
  [page 165] Strips leading or trailing characters
- `strip(String, Direction) -> Stripped`
  [page 165] Strips leading or trailing characters
- `strip(String, Direction, Character) -> Stripped`
  [page 165] Strips leading or trailing characters
- `left(String, Number) -> Left`
  [page 166] Adjusts left end of string
- `left(String, Number, Character) -> Left`
  [page 166] Adjusts left end of string
- `right(String, Number) -> Right`
  [page 166] Adjusts right end of string
- `right(String, Number, Character) -> Right`
  [page 166] Adjusts right end of string
- `centre(String, Number) -> Centered`
  [page 166] Centers a string
- `centre(String, Number, Character) -> Centered`
  [page 166] Centers a string
- `sub_string(String, Start) -> SubString`
  [page 166] Extracts a substring
- `sub_string(String, Start, Stop) -> SubString`
  [page 167] Extracts a substring

supervisor

The following functions are exported:

- `start_link(Module,StartArgs) -> SupRet`
  [page 168] Starts a supervisor process
- `start_link(SupName,Module,StartArgs) -> SupRet`
  [page 168] Starts a supervisor process
- `start_child(Supervisor,ChildSpec | ExtraStartArgs) -> {ok, Child} | {ok, Child, Info} | {error, Reason}`
  [page 169] Dynamically starts a child
- `terminate_child(Supervisor, Name) -> ok | {error, not_found}`
  [page 170] Terminates a child
- `delete_child(Supervisor,Name) -> ok | {error, running | not_found}`
  [page 170] Deletes a child from a supervisor
- `restart_child(Supervisor,Name) -> {ok, Pid} | {ok, Pid, Info} | {error, running | not_found | Reason}`
  [page 170] Starts a terminated child

- which_children(Supervisor) -> [{Name, Pid, Type, Modules}]
  [page 170] Gets the children of the supervisor
- check_childspecs([ChildSpec]) -> ok | {error, Reason}
  [page 171] Checks if a list of child specs are correct
- Module:init(StartArgs) -> {ok, {SupFlags, [ChildSpec]}} | ignore | {error, Reason}
  [page 171] Returns a supervisor specification

supervisor_bridge

The following functions are exported:

- start_link(Module,StartArgs) -> {ok, Pid} | ignore | {error, Reason}
  [page 173] Starts a supervisor bridge process
- start_link(Name,Module,StartArgs) -> {ok, Pid} | ignore | {error, Reason}
  [page 173] Starts a supervisor bridge process
- Module:init(StartArgs) -> {ok, Pid, State} | ignore | {error, Reason}
  [page 174] Initializes the supervisor bridge process
- Module:terminate(Reason, State) -> void()
  [page 174] Terminates the sub-system

sys

The following functions are exported:

- log(Name,Flag)
  [page 176] Logs system events in memory
- log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}
  [page 176] Logs system events in memory
- log_to_file(Name,Flag)
  [page 176] Logs system events to the specified file
- log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}
  [page 176] Logs system events to the specified file
- statistics(Name,Flag)
  [page 176]
- statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}
  [page 176]
- trace(Name,Flag)
  [page 177] Prints all system events on standard_io
- trace(Name,Flag,Timeout) -> void()
  [page 177] Prints all system events on standard_io
- no_debug(Name)
  [page 177] Turns off debugging

- `no_debug(Name,Timeout) -> void()`
  [page 177] Turns off debugging
- `suspend(Name)`
  [page 177] Suspends the process
- `suspend(Name,Timeout) -> void()`
  [page 177] Suspends the process
- `resume(Name)`
  [page 177] Resumes a suspended process
- `resume(Name,Timeout) -> void()`
  [page 177] Resumes a suspended process
- `change_code(Name, OldVsn, Module, Extra)`
  [page 177] Sends the code change system message to the process
- `change_code(Name, OldVsn, Module, Extra, Timeout) -> ok | {error, Reason}`
  [page 177] Sends the code change system message to the process
- `get_status(Name)`
  [page 177] Gets the status of the process
- `get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`
  [page 177] Gets the status of the process
- `install(Name,{Func,FuncState})`
  [page 178] Installs a debug function in the process
- `install(Name,{Func,FuncState},Timeout)`
  [page 178] Installs a debug function in the process
- `remove(Name,Func)`
  [page 178] Removes a debug function from the process
- `remove(Name,Func,Timeout) -> void()`
  [page 178] Removes a debug function from the process
- `debug_options(Options) -> [dbg_opt()]`
  [page 179] Converts a list of options to a debug structure
- `get_debug(Item,Debug,Default) -> term()`
  [page 179] Gets the data associated with a debug option
- `handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]`
  [page 179] Generates a system event
- `handle_system_msg(Msg,From,Parent,Module,Debug,Misc)`
  [page 179] Takes care of system messages
- `print_log(Debug) -> void()`
  [page 180] Prints the logged events in the debug structure
- `Mod:system_continue(Parent, Debug, Misc)`
  [page 180] Called when the process should continue its execution
- `Mod:system_terminate(Reason, Parent, Debug, Misc)`
  [page 180] Called when the process should terminate
- `Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}`
  [page 180] Called when the process should perform a code change

timer

The following functions are exported:

- `start() -> ok`
  [page 182] Starts a global timer server (named `timer_server`).
- `apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}`
  [page 182] Applies `Module:Function(Arguments)` after a specified `Time`.
- `send_after(Time, Pid, Message) -> {ok, TRef} | {error,Reason}`
  [page 182] Sends `Message` to `Pid` after a specified `Time`.
- `send_after(Time, Message) -> {ok, TRef} | {error,Reason}`
  [page 182] Sends `Message` to `Pid` after a specified `Time`.
- `exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}`
  [page 183] Send an exit signal with `Reason` after a specified `Time`.
- `exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}`
  [page 183] Send an exit signal with `Reason` after a specified `Time`.
- `kill_after(Time, Pid)-> {ok, TRef} | {error,Reason2}`
  [page 183] Send an exit signal with `Reason` after a specified `Time`.
- `kill_after(Time) -> {ok, TRef} | {error,Reason2}`
  [page 183] Send an exit signal with `Reason` after a specified `Time`.
- `apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`
  [page 183] Evaluates `Module:Function(Arguments)` repeatedly at intervals of `Time`.
- `send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 183] Sends `Message` repeatedly at intervals of `Time`.
- `send_interval(Time, Message) -> {ok, TRef} | {error, Reason}`
  [page 183] Sends `Message` repeatedly at intervals of `Time`.
- `cancel(TRef) -> {ok, cancel} | {error, Reason}`
  [page 183] Cancels a previously requested timeout identified by `TRef`.
- `sleep(Time) -> ok`
  [page 183] Suspends the calling process for `Time` amount of milliseconds.
- `tc(Module, Function, Arguments) -> {Time, Value}`
  [page 184] Measures the real time it takes to evaluate `apply(Module, Function, Arguments)`
- `seconds(Seconds) -> Milliseconds`
  [page 184] Converts `Seconds` to `Milliseconds`.
- `minutes(Minutes) -> Milliseconds`
  [page 184] Converts `Minutes` to `Milliseconds`.
- `hours(Hours) -> Milliseconds`
  [page 184] Converts `Hours` to `Milliseconds`.
- `hms(Hours, Minutes, Seconds) -> Milliseconds`
  [page 184] Converts `Hours+Minutes+Seconds` to `Milliseconds`.

unix

The following functions are exported:

- `cmd(String)`
  [page 186]

c (Module)

The `c` module enables users to enter the short form of some commonly used commands. These functions are are intended for interactive use in the Erlang shell.

Exports

`bt(Pid) -> void()`

Types:

- Pid = pid()

This function evaluates `erlang:process_display(Pid, backtrace)`.

`c(File) -> CompileResult`

This function is equivalent to:

`compile:file(File,[report_errors, report_warnings])`

`c(File, Flags) -> CompileResult`

Types:

- File = atom() | string()
- CompileResult = {ok, ModuleName} | error
- ModuleName = atom()
- Flags = [Flag]

This function calls the following function and then purges and loads the code for the file:

`compile:file(File, Flags ++ [report_errors, report_warnings])`

If the module corresponding to `File` is being interpreted, then `int:i` is called with the same arguments and the module is loaded into the interpreter. Note that `int:i` only recognizes a subset of the options recognized by `compile:file`.

Extreme care should be exercised when using this command to change running code which is executing. The expected result may not be obtained.

Refer to `compiler` manual pages for a description of the individual compiler flags.

`cd(Dir) -> void()`

Types:

- Dir = atom() | string()

This function changes the current working directory to `Dir`, and then prints the new working directory.

`flush() -> void()`

This function flushes all messages in the shell message queue.

`help() -> void()`

This function displays help about the shell and about the command interface module.

`i() -> void()`

This function provides information about the current state of the system. This call uses the BIFs `processes()` and `process_info/1` to examine the current state of the system. (The code is a good introduction to these two BIFs).

`zi() -> void()`

This function works like `i()`, but additionally displays information about zombie processes, i.e., processes which have exited, but which are still kept in the system to be inspected.

`ni() -> void()`

This function does the same as `i()`, but for all nodes in the network.

`i(X, Y, Z) -> void()`

Types:

- X = Y = Z = int()

This function evaluates `process_info(pid(X, Y, Z))`.

`l(Module) -> void()`

Types:

- Module = atom(), | string()

This function evaluates `code:purge(Module)` followed by `code:load_module(Module)`. It reloads the module.

`lc(ListOfFiles) -> Result`

Types:

- ListOfFiles = [File]
- File = atom() | string()
- Result = [CompileResult]
- CompileResult = {ok, ModuleName} | error
- ModuleName = atom()

This function compiles several files by calling `c(File)` for each file in `ListOfFiles`.

`ls() -> void()`

This function lists all files in the current directory.

`ls(Dir) -> void()`

Types:

- Dir = atom() | string()

This function lists all files in the directory `Dir`.

`m() -> void()`

This function lists the modules which have been loaded and the files from which they have been loaded.

`m(Module) -> void()`

Types:

- Module = atom()

This function lists information about `Module`.

`nc(File) -> void()`

Types:

- File = atom() | string()

This function compiles `File` and loads it on all nodes in an Erlang nodes network.

`nc(File, Flags) -> void()`

Types:

- File = atom() | string()
- Flags = [Flag]

This function compiles `File` with the additional compiler flags `Flags` and loads it on all nodes in an Erlang nodes network. Refer to the `compile` manual pages for a description of `Flags`.

`nl(Module) -> void()`

Types:

- Module = atom()

This function loads `Module` on all nodes in an Erlang nodes network.

`pid(X, Y, Z) -> pid()`

Types:

- X = Y = Z = int()

This function converts the integers `X`, `Y`, and `Z` to the Pid `<X.Y.Z>`. It saves typing and the use of `list_to_pid/1`. This function should only be used when debugging.

`pwd() -> void()`

This function prints the current working directory.

`q() -> void()`

This function is shorthand for `init:stop()`, i.e., it causes the node to stop in a controlled fashion.

`regs() -> void()`

This function displays formatted information about all registered processes in the system.

`nregs() -> void()`

This function is the same as `regs()`, but on all nodes in the system.

calendar (Module)

This module provides computation of local and universal time, day-of-the-week, and several time conversion functions.

Time is local when it is adjusted in accordance with the current time zone and daylight saving. Time is universal when it reflects the time at longitude zero, without any adjustment for daylight saving. Universal Coordinated Time (UTC) time is also called Greenwich Mean Time (GMT).

The time functions `local_time/0` and `universal_time/0` provided in this module both return date and time. The reason for this is that separate functions for date and time may result in a date/time combination which is displaced by 24 hours. This happens if one of the functions is called before midnight, and the other after midnight. This problem also applies to the Erlang BIFs `date/0` and `time/0`, and their use is strongly discouraged if a reliable date/time stamp is required.

All dates conform to the Gregorian calendar. This calendar was introduced by Pope Gregory XIII in 1582 and was used in all Catholic countries from this year. Protestant parts of Germany and the Netherlands adopted it in 1698, England followed in 1752, and Russia in 1918 (the October revolution of 1917 took place in November according to the Gregorian calendar).

The Gregorian calendar in this module is extended back to year 0. For a given date, the *gregorian days* is the number of days up to and including the date specified. Similarly, the *gregorian seconds* for a given date and time, is the the number of seconds up to and including the specified date and time.

For computing differences between epochs in time, use the functions counting gregorian days or seconds. If epochs are given as local time, they must be converted to universal time, in order to get the correct value of the elapsed time between epochs. Use of the function `time_difference/2` is discouraged.

Exports

`date_to_gregorian_days(Year, Month, Day) -> Days`
`date_to_gregorian_days(Date) -> Days`

Types:

- Date = {Year, Month, Day}
- Year = Month = Day = Days = int()

This function computes the number of gregorian days starting with year 0 and ending at the given date.

`datetime_to_gregorian_seconds(DateTime) -> Days`

Types:

- DateTime = {date(), time()}
- date() = {Year, Month, Day}
- time() = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = Days = int()

This function computes the number of gregorian seconds starting with year 0 and ending at the given date and time.

`day_of_the_week(Date) -> DayNumber`
`day_of_the_week(Year, Month, Day) -> DayNumber`

Types:

- Date = {Year, Month, Day}
- Year = Month = Day = DayNumber = int()

This function computes the day of the week given `Year`, `Month` and `Day`. The return value denotes the day of the week as follows:

`Monday = 1, Tuesday = 2, ..., Sunday = 7`

`Year` cannot be abbreviated and a value of 93 denotes the year 93, and not the year 1993. `Month` is the month number with January = 1. `Day` is an integer in the range 1 and the number of days in the month `Month` of the year `Year`.

`gregorian_days_to_date(Days) -> Date`

Types:

- Date = {Year, Month, Day}
- Year = Month = Day = Days = int()

This function computes the date given the number of gregorian days.

`gregorian_seconds_to_datetime(Secs) -> DateTime`

Types:

- DateTime = {date(), time()}
- date() = {Year, Month, Day}
- time() = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = Days = int()

This function computes the date and time from the given number of gregorian seconds.

`is_leap_year(Year) -> bool()`

Types:

- Year = int()

This function checks if a year is a leap year.

`last_day_of_the_month(Year, Month) -> int()`

Types:

- Year = Month = int()

This function computes the number of days in a month.

`local_time() -> {Date, Time}`

Types:

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

This function returns the local time reported by the underlying operating system.

`local_time_to_universal_time({Date, Time}) -> {Date, Time}`

Types:

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

This function converts from local time to Universal Coordinated Time (UTC). `Date` must refer to a local date after Jan 1, 1970.

`now_to_local_time(Now) -> {Date, Time}`

Types:

- Now = {MegaSecs, Secs, MicroSecs}
- Date = {Year, Month, Day}
- Time = {Hour, Minute, Second}
- MegaSecs = Secs = MilliSecs = int()
- Year = Month = Day = Hour = Minute = Second = int()

This function returns local date and time converted from the return value from `erlang:now()`.

`now_to_universal_time(Now) -> {Date, Time}`
`now_to_datetime(Now) -> {Date, Time}`

Types:

- Now = {MegaSecs, Secs, MicroSecs}
- Date = {Year, Month, Day}
- Time = {Hour, Minute, Second}
- MegaSecs = Secs = MilliSecs = int()
- Year = Month = Day = Hour = Minute = Second = int()

This function returns Universal Coordinated Time (UTC) converted from the return value from `erlang:now()`.

`seconds_to_daystime(Secs) -> {Days, Time}`

Types:

- Time() = {Hour, Minute, Second}
- Hour = Minute = Second = Days = int()

This function transforms a given number of seconds into days, hours, minutes, and seconds. The `Time` part is always non-negative, but `Days` is negative if the argument `Secs` is.

`seconds_to_time(Secs) -> Time`

Types:

- Time() = {Hour, Minute, Second}
- Hour = Minute = Second = Secs = int()

This function computes the time from the given number of seconds. `Secs` must be less than the number of seconds per day.

`time_difference(T1, T2) -> Tdiff`

Types:

- T1 = T2 = {Date, Time}
- Tdiff = {Day, {Hour, Minute, Second}}
- Date = {Year, Month, Day}
- Time = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()

This function returns the difference between two {`Date, Time`} structures. `T2` should refer to an epoch later than `T1`.

This function is obsolete. Use the conversion functions for gregorian days and seconds instead.

`time_to_seconds(Time) -> Secs`

Types:

- Time() = {Hour, Minute, Second}
- Hour = Minute = Second = Secs = int()

This function computes the number of seconds since midnight up to the specified time.

`universal_time() -> {Date, Time}`

Types:

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

This function returns the Universal Coordinated Time (UTC) reported by the underlying operating system. Local time is returned if universal time is not available.

`universal_time_to_local_time({Date, Time}) -> {Date, Time}`

Types:

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

This function converts from Universal Coordinated Time (UTC) to local time. `Date` must refer to a date after Jan 1, 1970.

```
valid_date(Date) -> bool()
valid_date(Year, Month, Day) -> bool()
```

Types:

- Date = {Year, Month, Day}
- Year = Month = Day = int()

This function checks if a date is a valid.

Leap Years

The notion that every fourth year is a leap year is not completely true. By the Gregorian rule, a year Y is a leap year if either of the following rules is valid:

- Y is divisible by 4, but not by 100; or
- Y is divisible by 400.

Accordingly, 1996 is a leap year, 1900 is not, but 2000 is.

Date and Time Source

Local time is obtained from the Erlang BIF `localtime/0`. Universal time is computed from the BIF `universaltime/0`.

The following facts apply:

- there are 86400 seconds in a day
- there are 365 days in an ordinary year
- there are 366 days in a leap year
- there are 1461 days in a 4 year period
- there are 36524 days in a 100 year period
- there are 146097 days in a 400 year period
- there are 719528 days between Jan 1, 0 and Jan 1, 1970.

dets (Module)

`dets` is a disk based version of the module `ets`. New users should read the documentation for the `ets` module before reading this description. In places where no description is given for the behavior of a function in this module, then the function behaves exactly as its corresponding function in the `ets` module.

This module provides a term (tuple) storage on file. It is possible to insert, delete, and search for specific terms in a file. The implementation is based on linear hashing. This module is used as the underlying file storage mechanism of the Mnesia DBMS. The module is provided as is, and without Mnesia, for users who are interested in an efficient storage of Erlang terms on disk only. Many applications only need to store some terms in a file. Mnesia adds transactions, queries, and distribution.

A file must be opened and closed. If a file has not been properly closed, the `dets` module will automatically repair the file. This might take some time if the file is very large. By default, files are closed if the process which opened the file terminates. If several Erlang processes open the same `dets` file, they will all share the file. The file is properly closed when all users has either terminated or closed the file. `dets` files are not properly closed if the Erlang system is terminated abnormally.

> **Note:**
> A ^C command abnormally terminates an Erlang runtime system in a Unix environment with a break-handler.

Since all operations in this module are disk operations, it is important to realize that a single look-up operation might involve a series of disk seek and read operations. For this reason, the operations in this module are much slower than the corresponding operation in `ets`, although this module exports a similar interface.

All functions in this module fail and return `{error, Reason}` if an error occurs.

The size of an empty `dets` file is approximately 34 kilobytes. This may seem large, but this is the price paid for searching for an object in an arbitrarily large file with almost constant search time.

The implementation of `dets` is based on the principle of the `ets` module. Data is organized as a linear hash list and the hash list grows gracefully the more data is inserted into the file. Space management on the file is performed by what is called a buddy system.

It is worth noting that the ordered_set data-type present in `ets` tables is not yet implemented in `dets`, neither is the limited support for concurrent updates which makes a `first/next` sequence safe to use on 'fixed' `ets` tables. Both these features will be implemented for `dets` in a future release of the Erlang/OTP system. Until then, the `Mnesia` DBMS (or some user implemented method for locking) has to be used to implement safe concurrency. No supplied library in Erlang/OTP currently has support for ordered disk based term storage.

Exports

`open_file(Name, Args) -> {ok, Name} | {error, Reason}`

This function opens a `dets` file. An empty `dets` file is created if no file exists.

The `Name` argument is the name of the table. The table name must be provided in all subsequent operations on the file. This means that `dets` files have atomic names. The name can be used by other processes as well, and several process can share one `dets` file. This behavior is similar to the `named_table` option in `ets`. If two processes open the same file, give the file the same name and provide the same arguments, then the file will have two users. If one user closes the file, it still remains open until the second user closes the file. The `Args` argument is a list of `{Key, Val}` tuple where the following values are allowed:

- `{type, Type}`, where `Type` must be either of the atoms `set`, `bag` or `duplicate_bag`. If a file is of type `set`, it means that each key uniquely identifies either one or zero objects. Thus, if a second object is inserted with a key that is already present in the file, then the first object will be overwritten. On the contrary, a file of type `bag` can have multiple objects with same key. However, identical instances of the same object cannot occur in the same file. If the type is set to `duplicate_bag` multiple identical objects may occur in the file. The default value is `set`.
- `{file, Filename}` is the name of the file to be opened. The default value is the name of the table.
- `{keypos, Pos}`. Only tuples can be inserted in a `dets` file. This attribute specifies which position in each tuple to use as the key field. The default value is `1`. The ability to change the key position is most convenient when we want to store Erlang records in which the first position of the tuple/record is the name of the record type.
- `{repair, Bool}` This flag specifies if the `dets` server invokes the automatic file repair algorithm. The default is `true`. If `false` is specified, there is no attempt to repair the file and the error `{error, need_repair}` is returned.
- `{cache_size, Integer}` The dets process can keep a cache of elements read (or written) to the file. The cache is “write-through”, i. e. the data is always saved to disk when inserting.

  The integer value is the number of keys kept in the cache, (the objects are also kept in the cache, but there can be more than one object per key in a `bag` or `duplicate bag`). The atom `infinity` can be supplied as `cache_size`, which indicates that the cache can grow infinitely (and be as large as the disk based table itself). A infinite cache may be an alternative to manually (or via Mnesia) shadowing a dets table in an ets ditto.

  Default is to have no cache at all (0).
- `{auto_save, Time}` If `auto_save` is specified, the dets table is flushed to disk whenever it’s not accessed for `Time` milliseconds. A dets table that is flushed will require no repair when reopened after an uncontrolled emulator halt.

  A `Time` value of `infinity` will disable auto save.

  The default value is 180000 (3 minutes).

- {ram_file, Bool} The dets file is kept in RAM memory if this flag is set. This may sound like an anomaly, but this flag can enhance the performance of applications which open a dets file, insert a set of objects, and then close the file. When the dets file is closed, its contents are written to the real disk file. The default value is false.
- {estimated_no_object, Int} Application performance can be enhanced with this flag by specifying, when the file is created, the estimated number of objects that will occupy the dets file. The default value as well as the minimum value is 256.
- {access, Access}. It is possible to open existing dets files in read-only mode. The value of the parameter Access is either read or read_write. The default value is read_write. A dets file which is opened in read-only mode is not marked as opened, and consequently it is not subjected to the automatic repair process if it is later opened.

The dets server keeps track of the number of users of each file. If a file is opened twice, it must be closed twice.

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

This function opens an existing dets file. If the file is not properly closed, it fails with {error, need_repair}. This function is most useful for debugging purposes.

close(Name) -> ok | {error, Reason}

This function closes a file. Only the owner of a dets file (i.e., the process which opened it) is allowed to close it.

All open files must be closed before the system is stopped. If we attempt to open a file which has not been properly closed, the dets module tries to automatically repair the file.

insert(Name, Object) -> ok | {error, Reason}

This function inserts an Object in table Name.

lookup(Name, Key) -> ObjectList | {error, Reason}

This function searches the table Name for object(s) with the key Key and returns a list of the found object(s). Insert and look-up times in tables are constant. For example:

```
2> dets:open_file(abc, [{type, bag}]).
{ok,abc}
3> dets:insert(abc, {1,2,3}).
ok
4> dets:insert(abc, {1,3,4}).
ok
5> dets:lookup(abc, 1).
[{1,2,3},{1,3,4}]
```

If the table is of type set, the function returns either [ ], or a list with a maximum length of one (there can be only be one object with a single key in a set). If the table is of type bag, a look-up returns a list of arbitrary length.

traverse(Name, Fun) -> Return

This function makes it possible to traverse a whole `dets` file and perform some operation on all or some objects in the file. Different actions are taken depending on the return value of `Fun`. The following `Fun` return values are allowed:

`continue` Continue to perform the traversal. For example, the following function is supplied in order to print the contents of a file to the terminal:

```
fun(X) -> io:format("~p~n", [X]), continue end.
```

`{continue, Val}` Continue the traversal *and* accumulate `Val`. The following function is supplied in order to collect all objects in a file into a list:

```
fun(X) -> {continue, X} end.
```

`{done, Value}` Terminate the search and return `[Value | Previously_accumulated]`.

`delete(Name, Key) -> ok`

This function deletes all objects with a specific key from a table.

`delete_object(Name, Object) -> ok`

This function deletes a specific object from a table. If a table is of type `bag`, the `delete/2` function cannot be used to delete only some of the objects with a specific key. This function makes this possible.

`first(Name) -> Key | '$end_of_table'`

This function returns the 'first' object in a table.

`next(Name, Key) -> Key | '$end_of_table'`

This function returns the next key in a table.

`slot(Name, I) -> $end_of_table | ObjList`

This function return the list of objects associated with slot `I`.

`all() -> NameList`

This function returns a list of all open files on this node.

`sync(Name) -> ok`

This function ensures that all data written to `Name` is written to disk. This also applies to files which have been opened with the `ram_file` flag set to `true`. In this case, the contents of the RAM file is flushed to disk.

`match_object(Name, Pattern) -> ObjectList`

This function matches objects and returns a list of all objects which match `Pattern`. If the keypos'th element of `Pattern` is unbound, a full search of file is performed. On the contrary, if the keypos'th element is not a variable, this function only searches among the objects with the right key.

`match(Name, Pattern) -> BindingsList`

This function matches objects and returns a list of all bindings which match `Pattern`. If the keypos'th element of `Pattern` is unbound, a full search over the whole file is performed. On the contrary, if the keypos'th element is not a variable, this function only searches among the objects with the right key.

`match_delete(Name, Pattern) -> ok`

Deletes all objects which matches `Pattern` from `Name`.

`info(Name) -> InfoList`

This function returns a list of {`Tag, Value`} pairs describing the file. The following list of items is returned.

- {`type, Type`}, where `Type` is either of the atoms `set` or `bag`.
- {`keypos, Pos`}.
- {`size, Size`}, where `Size` is the number of objects which reside in the file.
- {`file_size, Fz`}, where `Fz` is the size of the file in bytes.
- {`users, U`}. where `U` is list of the Pids which currently use the file.
- {`filename, F`}, where `F` is the name of the actual file being used.

`safe_fixtable(Name, true|false)`

This function works as the corresponding function in `ets`, except that it does *not* guarantee that `first/next` sequences during concurrent *deletes* work as expected. The limited support for concurrency implemented in `ets` tables is not yet implemented in `dets`. This interface currently only disables resizing of the hash area in a table. Until concurrent deletes are supported, the interface is of limited usage for others than the `Mnesia` DBMS. It is documented here for completeness.

`info(Name, Key) -> Value`

Returns one of the possible information fields which are available by means of `info/1`.

Additionally, the following `Keys` can be specified:

- `fixed`. Returns `true` if rehashing is disabled either by the `Mnesia` *internal* `fixtable/2` interface or by the `safe_fixtable/2` interface.
  The `Key` is special in that it returns the atom `undefined` if `Name` is not an open table. Other `Keys` will generate an exit signal (`badarg`) in the same situation, which is not compatible with `ets` and may be subject to change in future releases.
- `safe_fixed`. If the table is 'fixed' using `safe_fixtable/2`, the call returns a tuple: {`FixedNowTime,[{Pid,RefCount}]`}, where `FixedNowTime` is the time when the table was fixed by the first process (which may not be one of the processes fixing it now), `Pid` is a process 'fixing' the table right now and `RefCount` is the reference counter for 'fixes' done by that process. There may be any number of processes in the list. In *all* other cases, the atom `false` is returned.

See Also

ets(3), mnesia(3)

dict (Module)

`Dict` implements a `Key` - `Value` dictionary.

Exports

`append(Key, Value, Dict1) -> Dict2`

Types:

- Key = Value = term()
- Dict1 = Dict2 = dictionary()

This function appends a new `Value` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`append_list(Key, ValList, Dict1) -> Dict2`

Types:

- ValList = [Value]
- Key = Value = [term()]
- Dict1 = Dict2 = dictionary()

This function appends a list of values `ValList` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`dict_to_list(Dict) -> List`

Types:

- Dict = dictionary()
- List = [{Key, Value}]

This function converts the dictionary to a list representation.

`erase(Key, Dict1) -> Dict2`

Types:

- Key = term()
- Dict1 = Dict2 = dictionary()

This function erases all items with a given key from a dictionary.

`fetch(Key, Dict) -> Value`

Types:

- Key = Value = term()
- Dict = dictionary()

This function returns the value associated with `Key` in the dictionary `Dict`. `fetch` assumes that the `Key` is present in the dictionary and an exception is generated if `Key` is not in the dictionary.

`fetch_keys(Dict) -> Keys`

Types:

- Dict = dictionary()
- Keys = [term()]

This function returns a list of all keys in the dictionary.

`find(Key, Dict) -> Result`

Types:

- Key = term()
- Dict = dictionary()
- Result = {ok, Value} | error

This function searches for a key in a dictionary. Returns `{ok, Value}` where `Value` is the value associated with `Key`, or `error` if the key is not present in the dictionary.

`is_key(Key, Dict) -> bool()`

Types:

- Key = term()
- Dict = dictionary()

This function tests if `Key` is contained in the dictionary `Dict`

`new() -> dictionary()`

This function creates a new dictionary.

`store(Key, Value, Dict1) -> Dict2`

Types:

- Key = Value = term()
- Dict1 = Dict2 = dictionary()

This function stores a `Key` - `Value` pair in a dictionary. If the `Key` already exists in `Dict1`, the associated value is replaced by `Value`.

Notes

The functions `append` and `append_list` are included so we can store keyed values in a list *accumulator*. For example:

```
> D0 = dict:new(),
  D1 = dict:store(files, [], D0),
  D2 = dict:append(files, f1, D1),
  D3 = dict:append(files, f2, D2),
  D4 = dict:append(files, f3, D3),
  dict:fetch(files, D4).
[f1,f2,f3]
```

This saves the trouble of first fetching a keyed value, appending a new value to the list of stored values, and storing the result.

The function `fetch` should be used if the key is known to be in the dictionary, otherwise `find`.

digraph (Module)

This module implements directed graphs. A directed graph consists of a set of vertices (nodes) and a set of edges (connections). Both vertices and edges are identified with an Erlang term. It is possible to have multiple edges between vertices, and both vertices and edges may have user data attached.

Exports

`new(Opts) -> graph()`

Types:

- Opts = [cyclic|acyclic|public|private|protected]

Returns an empty graph of the type `Type`. `Type` is a list of type specifiers:

- `cyclic` Allows cyclic graphs.
- `acyclic` Does not allow cyclic graphs.
- `public` The graph may be read and modified by any process.
- `protected` Other processes can only read the graph.
- `private` The graph can be read and modified by the creating process.

`new() -> graph()`

Equivalent to the call `new([cyclic, protected])`.

`delete(G) -> true`

Types:

- G = graph()

Deletes the graph. This call is important because graphs are implemented with `ets`. There is no garbage collection of `ets` tables. The graph will, however, be deleted if the process that created the graph terminates.

`add_vertex(G, Vertex, Data) -> V`
`add_vertex(G, Vertex) -> V`
`add_vertex(G) -> V`

Types:

- G = graph()
- Vertex = term()

- Data = term()
- V = vertex()

add_vertex/3 creates or modifies the vertex Vertex with the associated data Data. Vertex and Data may be any Erlang term. Returns vertex V.

add_vertex/2 is equivalent to <c>add_vertex(G, Vertex, []).

add_vertex/1 adds a vertex with a generated name and the empty list as data. Returns the new vertex V.

`vertex(G, V) -> {V,Data} | false`

Types:

- G = graph()
- V = vertex()
- Data = term()

Returns {V, Data}, or false if the vertex V does not exist.

`vertices(G) -> Vertices`

Types:

- G = graph()
- Vertices = [vertex()]

Returns a list of all vertices in the graph G.

`del_vertex(G, V) -> true`

Types:

- G = graph()
- V = vertex()

Deletes the vertex V. The edges connected to vertex V are also deleted. Returns true.

`del_vertices(G, Vertices) -> true`

Types:

- G = graph()
- Vertices = [vertex()]

Deletes the vertices in the list Vertices. Returns true.

`add_edge(G, Edge, V1, V2, Data) -> E | {error, Reason}`
`add_edge(G, V1, V2, Data) -> E | {error, Reason}`
`add_edge(G, V1, V2) -> E | {error, Reason}`

Types:

- G = graph()
- Edge = term()
- V1 = V2 = vertex()
- Data = term()
- E = edge()
- Reason = {bad_edge, Path} | {bad_vertex, V}

- Path = [vertex()]

add_edge/5 adds a directed edge with the identifier Edge, to the start vertex V1 and end vertex V2. It also attaches Data to the edge Edge. Edge and Data may be any Erlang terms.

add_edge/4 works as add_edge/5, but a unique name is generated for the edge.

add_edge/3 is equivalent to add_edge(G,V1,V2,[]).

Returns:

- {error, {bad_edge, Path}} if adding the edge will create a cycle in an acyclic graph.
- {error, {bad_vertex, V}} if either end point does not exists in the graph.
- Edge E otherwise.

`del_edge(G, E) -> true`

Types:

- G = graph()
- E = edge()

Deletes the edge E. Returns true.

`del_edges(G, Edges) -> true`

Types:

- G = graph()
- Edges = [edge()]

Deletes the edges in the list Edges. Returns true.

`edge(G, E) -> {E, V1, V2, Data} | false`

Types:

- G = graph()
- E = edge()
- V1 = V2 = vertex()
- Data = term()

Returns {E, V1, V2, Data}, where V1 is the start vertex and V2 is the end vertex. edge/2 returns false if the edge E does not exist.

`edges(G) -> Edges`

Types:

- G = graph()
- Edges = [edge()]

Returns all edges of the graph G as a list Edges.

`out_neighbours(G, V) -> Vertices`

Types:

- G = graph()

- V = vertex()
- Vertices = [vertex()]

Returns a list of the vertices to which the vertex `V` is connected.

`in_neighbours(G, V) -> Vertices`

Types:

- G = graph()
- V = vertex()
- Vertices = [vertex()]

Returns a list of vertices connected to `V`

`out_edges(G, V) -> Edges`

Types:

- G = graph()
- V = vertex()
- Edges = [edge()]

Returns the list of edges with starting points in vertex `V`.

`in_edges(G, V) -> Edges`

Types:

- G = graph()
- V = vertex()
- Edges = [edge()]

Returns the list of edges with ending point in vertex `V`.

`edges(G, V) -> Edges`

Types:

- G = graph()
- V = vertex()
- Edges = [edge()]

Returns the list of all edges, both to and from vertex `V`.

`out_degree(G, V) -> int()`

Types:

- G = graph()
- V = vertex()

Returns the number of edges with starting points in vertex `V`.

`in_degree(G, V) -> int()`

Types:

- G= graph()
- V = vertex()

Returns the number of edges with ending points in vertex `V`.

`del_path(G, V1, V2) -> true`

Types:

- G = graph()
- V1 = V2 = vertex()

Deletes all paths from vertex `V1` to vertex `V2`. Returns `true`.

`get_path(G, V1, V2) -> Vertices | false`

Types:

- G = graph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

Finds a path from vertex `V1` to vertex `V2`. Returns the path as the list `[V1, ..., V2]`, or `false` if no path exists.

`get_cycle(G, V) -> Vertices | false`

Types:

- G = graph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

Finds a cycle through vertex `V`. It first attempts to find cycles longer than one, and then a cycle of one. Returns the cycle as `[V, ..., V]` for lengths greater then one, `[V]` for lengths of one, and `false` if no cycle is found.

epp (Module)

The Erlang code preprocessor includes functions which are used by `compile` to preprocess macros and include files before the actual parsing takes place.

Exports

`open(FileName, IncludePath) -> {ok,Epp} | {error, ErrorDescriptor}`
`open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}`

Types:

- FileName = atom() | string()
- IncludePath = [DirectoryName]
- DirectoryName = atom() | string()
- PredefMacros = [{atom(),term()}]
- Epp = pid() – handle to the epp server
- ErrorDescriptor = term()

Opens a file for preprocessing.

`close(Epp) -> ok`

Types:

- Epp = pid() – handle to the epp server

Closes the preprocessing of a file.

`parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`

Types:

- Epp = pid()
- AbsForm = term()
- Line = integer()
- ErrorInfo = see separate description below.

Returns the next Erlang form from the opened Erlang source file. The tuple `{eof, Line}` is returned at end-of-file. The first form corresponds to an implicit attribute `-file(File,1).`, where `File` is the name of the file.

`parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}`

Types:

- FileName = atom() | string()

- IncludePath = [DirectoryName]
- DirectoryName = atom() | string()
- PredefMacros = [{atom(),term()}]
- Form = term() – same as returned by erl_parse:parse_form

Preprocesses and parses an Erlang source file. Note that the tuple `{eof, Line}` returned at end-of-file is included as a "form".

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

erl_parse [page 66]

erl_eval (Module)

This module provides an interpreter for Erlang expressions. The expressions are in the abstract syntax as returned by `erl_parse`, the Erlang parser, or a call to `io:parse_erl_exprs/2`.

Exports

`exprs(Expressions, Bindings) -> {value, Value, NewBindings}`
`exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`

Types:

- Expressions = as returned by erl_parse or io:parse_erl_exprs/2
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none

Evaluates `Expressions` with the set of bindings `Bindings`, where `Expressions` is a sequence of expressions (in abstract syntax) of a type which may be returned by `io:parse_erl_exprs/2`. See below for an explanation of how and when to use the argument `LocalFunctionHandler`.

Returns {`value, Value, NewBindings`}

`expr(Expression, Bindings) -> { value, Value, NewBindings }`
`expr(Expression, Bindings, LocalFunctionHandler) -> { value, Value, NewBindings }`

Types:

- Expression = as returned by io:parse_erl_form/2, for example
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none

Evaluates `Expression` with the set of bindings `Bindings`. `Expression` is an expression (in abstract syntax) of a type which may be returned by `io:parse_erl_form/2`. See below for an explanation of how and when to use the argument `LocalFunctionHandler`.

Returns {`value, Value, NewBindings`}.

`expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
`expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`

Evaluates a list of expressions in parallel, using the same initial bindings for each expression. Attempts are made to merge the bindings returned from each evaluation. This function is useful in the `LocalFunctionHandler`. See below.

Returns {`ValueList, NewBindings`}.

`new_bindings() -> BindingStruct`

Returns an empty binding structure.

`bindings(BindingStruct) -> Bindings`

Returns the list of bindings contained in the binding structure.

`binding(Name, BindingStruct) -> Binding`

Returns the binding of `Name` in `BindingStruct`.

`add_binding(Name, Value, Bindings) -> BindingStruct`

Adds the binding `Name = Value` to `Bindings`. Returns an updated binding structure.

`del_binding(Name, Bindings) -> BindingStruct`

Removes the binding of `Name` in `Bindings`. Returns an updated binding structure.

Local Function Handler

During evaluation of a function, no calls can be made to local functions. An undefined function error would be generated. However, the optional argument `LocalFunctionHandler` may be used to define a function which is called when there is a call to a local function. The argument can have the following formats:

{`value,Func`} This defines a local function handler which is called with:

`Func(Name, Arguments)`

`Name` is the name of the local function and `Arguments` is a list of the *evaluated* arguments. The function handler returns the value of the local function. In this case, it is not possible to access the current bindings. To signal an error, the function handler just calls `exit/1` with a suitable exit value.

{`eval,Func`} This defines a local function handler which is called with:

`Func(Name, Arguments, Bindings)`

`Name` is the name of the local function, `Arguments` is a list of the *unevaluated* arguments, and `Bindings` are the current variable bindings. The function handler returns:

`{value,Value,NewBindings}`

`Value` is the value of the local function and `NewBindings` are the updated variable bindings. In this case, the function handler must itself evaluate all the function arguments and manage the bindings. To signal an error, the function handler just calls `exit/1` with a suitable exit value.

`none` There is no local function handler.

Bugs

The evaluator is not complete. `receive` cannot be handled properly.

Any undocumented functions in `erl_eval` should not be used.

erl_id_trans (Module)

This module performs an identity parse transformation of Erlang code. It is included as an example for users who may wish to write their own parse transformers. If the option {`parse_transform,Module`} is passed to the compiler, a user written function `parse_transform/2` is called by the compiler before the code is checked for errors.

Exports

`parse_transform(Forms, Options) -> Forms`

Types:

- Forms = [erlang_form()]
- Options = [compiler_options()]

Performs an identity transformation on Erlang forms, as an example.

Parse Transformations

Parse transformations are used if a programmer wants to use Erlang syntax, but with different semantics. The original Erlang code is then transformed into other Erlang code.

> Note:
> Programmers are strongly advised not to engage in parse transformations and no support is offered for problems encountered.

See Also

erl_parse [page 66] compile.

erl_internal (Module)

This module defines Erlang BIFs, guard tests and operators. This module is only of interest to programmers who manipulate Erlang code.

Exports

`bif(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is an Erlang BIF which is automatically recognized by the compiler, otherwise `false`.

`guard_bif(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is an Erlang BIF which is allowed in guards, otherwise `false`.

`type_test(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is a valid Erlang type test, otherwise `false`.

`arith_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is an arithmetic operator, otherwise `false`.

`bool_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a Boolean operator, otherwise `false`.

`comp_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a comparison operator, otherwise `false`.

`list_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a list operator, otherwise `false`.

`send_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a send operator, otherwise `false`.

`op_type(OpName, Arity) -> Type`

Types:

- OpName = atom()
- Arity = integer()
- Type = arith | bool | comp | list | send

Returns the `Type` of operator that `OpName/Arity` belongs to, or generates a `function_clause` error if it is not an operator at all.

erl_lint (Module)

This module is used to check Erlang code for illegal syntax and other bugs. It also warns against coding practices which are not recommended.

The errors detected include:

- redefined and undefined functions
- unbound and unsafe variables
- illegal record usage.

Warnings include:

- unused functions and imports
- variables imported into matches
- variables exported from `if`/`case`/`receive`
- variables shadowed in lambdas and list comprehensions.

Some of the warnings are optional, and can be turned on by giving the appropriate option, described below.

The functions in this module are invoked automatically by the Erlang compiler and there is no reason to invoke these functions separately unless you have written your own Erlang compiler.

Exports

```
module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}
```

Types:

- AbsForms = [term()]
- FileName = FileName2 = atom() | string()
- Warnings = Errors = [{Filename2,[ErrorInfo]}]
- ErrorInfo = see separate description below.
- CompileOptions = [term()]

This function checks all the forms in a module for errors. It returns:

`{ok,Warnings}` There were no errors in the module.

`{error,Errors,Warnings}` There were errors in the module.

The elements of `Options` selecting optional warnings are as follows:

`{format, Verbosity}` Causes warnings to be emitted for malformed format strings as arguments to `io:format` and similar functions. `Verbosity` selects the amount of warnings: 0 = no warnings; 1 = warnings for invalid format strings; 2 = warnings also when the validity couldn't be checked (for example, when the format string argument is a variable).

`unused_vars` Causes warnings to be emitted for variables which are not used, with the exception of variables beginning with an underscore ("Prolog style warnings").

The `AbsForms` of a module which comes from a file that is read through `epp`, the Erlang pre-processor, can come from many files. This means that any references to errors must include the file name (see epp [page 55], or parser erl_parse [page 66] The warnings and errors returned have the following format:

```
[{FileName2,[ErrorInfo]}]
```

The errors and warnings are listed in the order in which they are encountered in the forms. This means that the errors from one file may be split into different entries in the list of errors.

`is_guard_test(Expr) -> bool()`

Types:

- Expr = term()

This function tests if `Expr` is a legal guard test. `Expr` is an Erlang term representing the abstract form for the expression. `erl_parse:parse_exprs(Tokens)` can be used to generate a list of `Expr`.

`format_error(ErrorDescriptor) -> string()`

Types:

- ErrorDescriptor = errordesc()

Takes an `ErrorDescriptor` and returns a string which describes the error or warning. This function is usually called implicitly when processing an `ErrorInfo` structure (see below).

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

erl_parse [page 66], epp [page 55]

erl_parse (Module)

This module is the basic Erlang parser which converts tokens into the abstract form of either forms (i.e., top-level constructs), expressions, or terms. Note that a token list must end with the *dot* token in order to be acceptable to the parse functions (see erl_scan).

Exports

`parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsForm = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a form. It returns:

`{ok, AbsForm}` The parsing was successful. See section Abstract Form [page 68] below for a description of `AbsForm`.

`{error, ErrorInfo}` An error occurred.

`parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Expr_list = [AbsExpr]
- AbsExpr = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a list of expressions. It returns:

`{ok, Expr_list}` The parsing was successful. `Expr_list` is a list of the form `AbsExpr`, which is described in the section Abstract Form [page 68] below.

`{error, ErrorInfo}` An error occurred.

`parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Term = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a term. It returns:

`{ok, Term}` The parsing was successful. `Term` is the Erlang term corresponding to the token list.

`{error, ErrorInfo}` An error occurred.

`format_error(ErrorDescriptor) -> string()`

Types:

- ErrorDescriptor = errordesc()

Uses an `ErrorDescriptor` and returns a string which describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed (see below).

`tokens(AbsTerm) -> Tokens`
`tokens(AbsTerm, MoreTokens) -> Tokens`

Types:

- Tokens = MoreTokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsTerm = term()
- ErrorInfo = see section Error Information below.

This function generates a list of tokens representing the abstract form `AbsTerm` of an expression. Optionally, it appends `Moretokens`.

`normalise(AbsTerm) -> Data`

Types:

- AbsTerm = Data = term()

Converts the abstract form `AbsTerm` of a term into a conventional Erlang data structure (i.e., the term itself). This is the inverse of `abstract/1`.

`abstract(Data) -> AbsTerm`

Types:

- Data = AbsTerm = term()

Converts the Erlang data structure `Data` into an abstract form of type `AbsTerm`. This is the inverse of `normalise/1`.

Abstract Form

To be supplied

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

io [page 112], erl_scan [page 72]

erl_pp (Module)

The functions in this module are used to generate aesthetically attractive representations of abstract forms, which are suitable for printing. All functions return (possibly deep) lists of characters and generate an error if the form is wrong.

All functions can have an optional argument which specifies a hook that is called if an attempt is made to print an unknown form.

Exports

```
form(Form) -> DeepCharList
form(Form, HookFunction) -> DeepCharList
```

Types:

- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

Pretty prints a `Form` which is an abstract form of a type which is returned by `erl_parse:parse_form`.

```
attribute(Attribute) -> DeepCharList
attribute(Attribute, HookFunction) -> DeepCharList
```

Types:

- Attribute = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the attribute `Attribute`.

```
function(Function) -> DeepCharList
function(Function, HookFunction) -> DeepCharList
```

Types:

- Function = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the function `Function`.

```
guard(Guard) -> DeepCharList
```

`guard(Guard, HookFunction) -> DeepCharList`

Types:

- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the guard test `Guard`.

`exprs(Expressions) -> DeepCharList`
`exprs(Expressions, HookFunction) -> DeepCharList`
`exprs(Expressions, Indent, HookFunction) -> DeepCharList`

Types:

- Expressions = term()
- HookFunction = see separate description below.
- Indent = integer()
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the sequence of expressions in `Expressions`.

`expr(Expression) -> DeepCharList`
`expr(Expression, HookFunction) -> DeepCharList`
`expr(Expression, Indent, HookFunction) -> DeepCharList`
`expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList`

Types:

- Expression = term()
- HookFunction = see separate description below.
- Indent = integer()
- Precedence =
- DeepCharList = [char()|DeepCharList]

This function prints one expression. It is useful for implementing hooks (see below).

Unknown Expression Hooks

The optional argument `HookFunction`, shown in the functions described above, defines a function which is called when an unknown form occurs where there should be a valid expression. It can have the following formats:

`Function` The hook function is called by:

```
Function(Expr,
         CurrentIndentation,
         CurrentPrecedence,
         HookFunction)
```

`none` There is no hook function

The called hook function should return a (possibly deep) list of characters. `expr/4` is useful in a hook.

If `CurrentIndentation` is negative, there will be no line breaks and only a space is used as a separator.

Bugs

It should be possible to have hook functions for unknown forms at places other than expressions.

See Also

io [page 112], erl_parse [page 66], erl_eval [page 57]

erl_scan (Module)

This module contains functions for tokenizing characters into Erlang tokens.

Exports

`string(CharList,StartLine]) -> {ok, Tokens, EndLine} | Error`
`string(CharList) -> {ok, Tokens, EndLine} | Error`

Types:

- CharList = string()
- StartLine = EndLine = Line = integer()
- Tokens = [{atom(),Line}|{atom(),Line,term()}]
- Error = {error, ErrorInfo, EndLine}

Takes the list of characters `CharList` and tries to scan (tokenize) them. Returns `{ok, Tokens, EndLine}`, where `Tokens` are the Erlang tokens from `CharList`. `EndLine` is the last line where a token was found.

`StartLine` indicates the initial line when scanning starts. `string/1` is equivalent to `string(CharList,1)`.

`{error, ErrorInfo, EndLine}` is returned if an error occurs. `EndLine` indicates where the error occurred.

`tokens(Continuation, CharList, StartLine) ->Return`

Types:

- Return = {done, Result, LeftOverChars} | {more, Continuation}
- Continuation = [] | string()
- CharList = string()
- StartLine = EndLine = integer()
- Result = {ok, Tokens, EndLine} | {eof, EndLine}
- Tokens = [{atom(),Line}|{atom(),Line,term()}]

This is the re-entrant scanner which scans characters until a *dot* ('.' whitespace) has been reached. It returns:

`{done, Result, LeftOverChars}` This return indicates that there is sufficient input data to get an input. `Result` is:

`{ok, Tokens, EndLine}` The scanning was successful. `Tokens` is the list of tokens including *dot*.

`{eof, EndLine}` End of file was encountered before any more tokens.

{error, ErrorInfo, EndLine} An error occurred.

{more, Continuation} More data is required for building a term. Continuation must be passed in a new call to tokens/3 when more data is available.

reserved_word(Atom) -> bool()

Returns true if Atom is an Erlang reserved word, otherwise false.

format_error(ErrorDescriptor) -> string()

Types:

- ErrorDescriptor = errordesc()

Takes an ErrorDescriptor and returns a string which describes the error or warning. This function is usually called implicitly when processing an ErrorInfo structure (see below).

Error Information

The ErrorInfo mentioned above is the standard ErrorInfo structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

Notes

The continuation of the first call to the re-entrant input functions must be []. Refer to Armstrong, Virding and Williams, 'Concurrent Programming in Erlang', Chapter 13, for a complete description of how the re-entrant input scheme works.

See Also

io [page 112] erl_parse [page 66]

ets (Module)

This module acts as an interface to the Erlang built-in term storage BIFs. The module provides the ability to store very large quantities of data in an Erlang system, and to have `constant` access time to this data (or in the case of the `ordered_set` data-type access time proportional to the logarithm of the number of elements in the table). Data is organized as a set of dynamic tables. Each table is created by a process. When the process terminates, the table is automatically destroyed. A table can store tuples. Every table has access rights set at creation.

The number of tables stored on one Erlang node is limited. The current default limit is approximately 1400 tables. The upper limit can be increased by setting the environment variable `ERL_MAX_ETS_TABLES` before starting the Erlang system (i.e. with the `-env` option to `erl`/`werl`). The actual limit may be slightly higher than the one specified, but never lower.

Tables are divided into four different types, `set`, `ordered_set`, `bag` and `duplicate_bag`. A `set` or `ordered_set` table can only have one tuple associated with each key, a `bag` table can have multiple tuples associated with a single key whereas a `duplicate_bag` table can have multiple identical objects in the same table.

In the current implementation, every object insert and look-up operation results in one copy of the object.

This module provides very limited support for concurrent updates. No locking is available, but the `safe_fixtable/2` function can be used to guarantee that a sequence of `first/1` and `next/2` calls will traverse the table without errors even if another process (or the same process) simultaneously deletes or inserts elements in the table.

If desired, locking and transactions must be implemented on top of these functions. This is done by the mnesia database system.

There is no automatic garbage collection for tables. The table is not destroyed automatically if there are no references to it from a process. The table has to be destroyed explicitly at user level. It is destroyed if the owner terminates, or with `delete/1`.

`'$end_of_table'` should not be used as a key since this atom is used to mark the end of the table when using first/next.

In general, the functions will exit with reason `badarg` if any argument is of the wrong format, or if the table ID is invalid.

> **Warning:**
> In the next release of OTP, R6, support for using ets tables on other nodes will be dropped. At the same time, the representation of table identifiers will be changed and the internal undocumented BIFs that the `ets` module used to call will be removed. These changes will result in faster access to ets tables.

Exports

`new(Name, Type)`

Creates a new table and returns a table identifier which can be used in subsequent operations. This table ID can also be sent to other processes so that a table can be shared between processes. It is completely location transparent and can be sent to processes at other nodes. Accordingly, the table identifier can be used as a location transparent store. Large amounts of data can be distributed to locations where it can be stored.

The parameter `Type` is a list which defaults to `[set, protected]` if `[]` is specified. The list may contain the following atoms:

- `set` The table is a `set` table - one key, one object, no order among elements.
- `ordered_set` The table is a `ordered_set` table - one key, one object, ordered in Erlang term order, which is the order implied by the < and > operators. Tables of this type behave slightly differently in some situations. Each API function of concern notes this different behaviour.
- `bag` The table is a `bag` table which can have multiple objects per key.
- `duplicate_bag` The table is a `duplicate_bag` table which can have multiple copies of the same object.
- `public` The table is open to both read and write operations. Any process may read or write to the table. If this option is used, the `ets` table can be seen as a shared memory segment which is shared by all Erlang processes.
- `protected` The owner can read and write to the table. Other processes can only read the table.
- `private` Only the owner process can read or write to the table.
- `named_table` If this option is present, the table can be accessed by name. With this option, it is possible to have globally accessible tables without passing the table identifier around.
- `{keypos, Pos}` By default, the first element of each tuple inserted in a table is the key. However, this might not always be appropriate. In particular, we do not want the first element to be the key if we want to insert Erlang records in a table. When creating a table, it is possible to specify which tuple position is the key.

> **Warning:**
> Do not assume anything about the datatype of the table identfier. In particular, the data type will be changed in release R6 of OTP.

`insert(Tab, Object)`

Inserts `Object` into the table `Tab`. The object must be a tuple with a size equal to or greater than one. If the table was created with the `keypos` option, the size can also be supplied there. By default, the first element of the object is the key of the object. Returns `true`.

`lookup(Tab, Key)`

Searches the table `Tab` for object(s) with the key `Key` and returns a list of the found object(s). Insert and look-up times in tables of type `set`, `bag` and `duplicate_bag` are constant, regardless of the size of the table. For the `ordered_set` data-type, the look-up time is proportional to the (binary) logarithm of the number of elements (it is implemented as a tree).

The following example illustrates:

```
1>T=ets:new(mytab, [bag, public]).
{6, <0.19.0>}
2> ets:insert(T, {a, 2, xx, yy}).
true
3> ets:insert(T, {a, 2, {peter, pan}, 77}).
true
4> ets:lookup(T, a).
[{a, 2, xx, yy}, {a, 2, {peter, pan}, 77}]
5> ets:insert(T, {b, 123, {peter,  pan}, 77}).
true
6> ets:lookup(T, b).
[{b, 123, {peter, pan}, 77}]
```

If the table is of type `set` or `ordered_set`, the function returns either `[]`, or a list of maximum length of one (there can be only be one object with a single key in a set).

If the table is of type `bag` or `duplicate_bag` , a look-up returns a list of arbitrary length. It is also worthwhile to note that `bag` tables have the following two properties.

- The same object cannot occur twice in the same table (no duplicates).
- The time order of object insertions is preserved. If object `{x, X}` is inserted before object `{x, Y}`, the call `ets:lookup(T, x)` is guaranteed to return the list `[{x, X}, {x, Y}]`, as opposed to the list `[{x, Y}, {x, X}]`

`lookup_element(Tab, Key, Pos)`

This function looks up the Pos'th element of the object in table `Tab`, with key `Key`. If no such object exists, the function exists with reason `badarg`. If the table is of type `bag` or `duplicate_bag`, a list of the elements is returned.

`delete(Tab, Key) -> true`

Deletes object(s) with the key `Key` in the table `Tab`. Returns `true`, or exits with reason `badarg` if `Tab` is not a valid Table.

`delete(Tab)`

Deletes the table `Tab`. Returns `true`, or exits with reason `badarg` if `Tab` is not a valid Table.

`update_counter(Tab, Key, Incr)`

In a table of type `set` or `ordered_set`, an efficient way of managing counters is to use an object with one or more integers to associate one or more counters with `Key`. The function `update_counter/3` destructively changes the object with key `Key` by adding the integer value `Incr` to the counter. The return value is the new value of the counter. `Incr` can be either:

- An integer that is added to the (integer) element directly following the key in the tuple (i.e. at position <keypos> + 1)
- A tuple `{Pos, Increment}` where `Pos` is the position of the counter element in the tuple and `Increment` is the integer value to be added to that element.

This function fails with `badarg` if:

- no object with the right key exists
- the object in the counter position is not an integer
- the table is of type `duplicate_bag` or `bag`
- the object in the table has the wrong arity.

`first(Tab)`

Returns the 'first' `Key` in the table `Tab`. There is no apparent order among the objects in tables of other types than `ordered_set`, but there is always an internal order known only by the table itself. In the case of the `ordered_set` table type, the first key in Erlang term order is returned. Returns `'$end_of_table'` if there is no first key (the table is empty).

`next(Tab, Key)`

Returns the 'next' table key after `Key`. `'$end_of_table'` is returned if the object associated with `Key` is the 'last' object in the table. As with `first/1` the only table type where the order has a meaning is `ordered_set`. For the table types `set`, `bag` and `duplicate_bag` the function fails with `badarg` if there is no object with the key `Key`, except for the case when the object with the associated key has been deleted from a (still) *fixed* table (see `safe_fixtable/2` below). If the table is of type `ordered_set` the function returns the next object in order, disregarding the fact that the key `Key` may or may not exist.

`last(Tab)`

Works exactly as `first/1` but returns the last object in Erlang term order for the `ordered_set` table type. For all other table types, `first/1` and `last/1` are synonyms.

`prev(Tab, Key)`

Returns the previous table key, which only has meaning for the `ordered_set` table type. For all other table types, `next/2` and `prev/2` are synonyms, one cannot backup to an 'object passed earlier' in a table of other type than `ordered_set`.

`slot(Tab, I)`

This is another way of traversing a table. The first `slot` of a table is 0 and the table can be traversed with consecutive calls to `slot/2`. Each call returns a list of objects. `'$end_of_table'` is returned when the end of the table is reached. This function fails with `badarg` if the `I` argument is out of range.

While consecutive calls to `slot` may look like a safe way to traverse a table even if it is concurrently updated by another process, it *is not so*. A sequence of calls to `slot/2` may result in unexpected `badarg`'s if the table is internally resized as an effect of deletes made from another process (or the traversing process itself). By using `safe_fixtable/2`, the table will not resize, but then again a sequence of `first/1` and `next/2` can be used safely on a `fixed` table, so `slot` is not safer than `first/1` and `next/2`.

For the `ordered_set` data-type, this function has even more limited usage. It will return a list containing the `I`:th element in the table (in Erlang term order). Concurrent updates can make a traversal of an `ordered_set` using `slot/2` behave very unexpectedly. Calls to `slot/2` on `ordered_set`'s with the index given (`I`) equal to the number of objects in the table will return the atom '$end_of_table'. Calls with indexes larger than the number of elements will result in a `badarg` exit.

Do not use this function. It may be removed in a future release.

`fixtable(Tab, true|false)`

This function toggles the table ability to "rehash" itself. It is primarily used by the Mnesia DBMS to implement functions which allow write operations in a table, although the table is in the process of being copied to disk or to another node.

The function keeps no track of when and how tables are fixed, it is actually more to be regarded as an internal interface used from the `safe_fixtable/2` function. It is retained *only* for backward compatibility, use `safe_fixtable/2` instead.

`safe_fixtable(Tab, true|false)`

This function implements limited concurrency support for tables of the `set`, `bag` and `duplicate_bag` table types. When a process 'fixes' a table, it remains `fixed` until that process either 'releases' the table or the process dies. If several processes 'fixes' a table, the table will be released when the *last* process releases it (or exits). A reference counter is also kept on a per process basis, so `N` consecutive 'fixes' of a table requires `N` 'releases' to actually release the table.

When a table is 'fixed', a sequence of `first/1` and `next/2` calls are guaranteed to succeed, that is without generating exits due to deleted keys used in the `next/2` call. An example follows:

```
clean_all_with_value(Tab, X) ->
  safe_fixtable(Tab, true), % Make sure the table is
                            % not rehashed.
  clean_all_with_value(Tab,X,ets:first(Tab)),
  safe_fixtable(Tab,false).

clean_all_with_value(Tab,X,'$end_of_table') ->
  true;
clean_all_with_value(Tab,X,Key) ->
  case ets:lookup(Tab,Key) of
    [{Key,X}] ->
      ets:delete(Tab,Key);
    _ -> % This may be either [{Key,_}] or [] due to
```

```
                % concurrent updates
            true
        end,
        clean_all_with_value(Tab,X,ets:next(Tab,Key)).
```

The above example would have generated a `badarg` exit if the table had not been 'fixed' before the loop `clean_all_with_value/3`.

Note that a table which is 'fixed' does not actually remove the elements deleted until it is 'released' by all processes that have 'fixed' it. If a process 'fixes' the table and *never* releases it, the memory used by the deleted objects will never be freed. The performance of operations on the table will also degrade significantly.

By using calls to `info/2`, one can inspect which processes are 'fixing' the table and when it was first 'fixed'. A system where a lot of processes are 'fixing' tables may need a process that monitors those tables and sends alarms when tables have been 'fixed' for to long.

For tables of the `ordered_set` type, 'fixing' has no usage, consecutive calls to `first/1` and `next/2` will always succeed, regardless of if the table is 'fixed' or not.

`all()`

Returns a list of all tables on this node.

`match(Tab, Pattern)`

Tries to match the object(s) in table `Tab` with the pattern `Pattern`. `Pattern` may contain '_' , which matches any object, bound parts, and variables. Pattern variables have the form of atoms beginning with a '$' sign and followed by a number, e.g., `'$0'` or `'$31'`. If successful, the result of the call is a list of variable bindings. The reason for providing a matching function is to scan large portions of a table, searching for a particular object without having to copy the entire table from the table space to the user space.

The following interaction with the Erlang shell illustrates how to use the `match/2` function:

```
7> ets:match(T, {a, 2, '$1', '$2'}).
 [[{peter, pan}, 77], [xx, yy]]
```

The call to `match/2` returned an ordered list of the variable bindings which is the first object that matched the pattern, bound the variable `$1` to `{peter, pan}`, and the variable `$2` to 77. The second object which matched the pattern bound the variable `$1` to `xx`, and the variable `$2` to `yy`. The pattern '_' can be used as a wild-card. It matches everything, but it does not bind any variables.

```
8> ets:match(T, {a, 2, '$1', '_'}).
 [[{peter, pan}], [xx]]
```

`[]` is returned if no match is found.

The first part of the objects are used as keys in the tables and a match request with the first part of the bound pattern - not a variable or an underscore - is very efficient. However, if the key part of the pattern is a variable, the entire table must be searched. The search time can be substantial if the table is very large.

The special case where the pattern is a single variable will collect the entire table.

```
9> ets:match(T, '$1').
 [[{a, 2, {peter, pan}, 77}], [{a, 2, xx, yy}],
  [{b, 123, {peter, pan}, 77}]]
```

On tables of the `ordered_set` data-type, the result is in the same order as in a `first/1`, `next/2` sequence.

`match_object(Tab, Pattern)`

Tries to match the object(s) in table `Tab` with the pattern `Pattern`. `Pattern` may contain '_' , which matches any object, bound parts, and variables. Pattern variables have the form of atoms beginning with a '$' sign and followed by a number, e.g., `'$0'` or `'$31'`. The result is a list of matching objects (i.e complete table objects). This function differs from `match/2` in that it returns complete objects and does not return any variable bindings. It is thus not very meaningful to use pattern variables, it will have exactly the same effect as using `'_'`.

The following interaction with the Erlang shell illustrates how to use the `match_object/2` function:

```
7> ets:match(T, {a, 2, '$1', '$2'}).
 [{a, 2, peter, pan}, {a, 2, captain, hook}]
```

The call to `match_object/2` returned an ordered list of objects that matched the pattern,

`[]` is returned if no match is found.

The first part of the objects are used as keys in the tables and a match request with the first part of the bound pattern - not a variable or an underscore - is very efficient. However, if the key part of the pattern is a variable, the entire table must be searched. The search time can be substantial if the table is very large.

The special case where the pattern is a single variable or `'_'` will collect the entire table.

On tables of the `ordered_set` data-type, the result is in the same order as in a `first/1`, `next/2` sequence.

`match_delete(Tab, Pattern)`

Deletes object(s) which match `Pattern` in the table `Tab`. This can be especially useful in combination with `bag` type tables. If the first element of `Pattern` is a variable, the entire table must be searched. Returns `true`.

`rename(Tab,NewName)`

Renames a (preferably) named table to the name `NewName`. `NewName` has to be an atom. Renaming a table that is not named will succeed, but is of course quite useless. The old name of a named table can no longer be used to access it after it is renamed.

`info(Tab)`

Returns a tagged structure which describes the table with the following tags:

- `memory` The number of words allocated to the table.
- `owner` The Pid of the owner of the table.
- `size` The number of objects inserted in the table.

- `type` Type `bag`, `duplicate_bag` or type `set`.
- `protection` Public, protected, or private.
- `node` The name of the node where `Tab` is actually stored.
- `name` The name of the table, as given to `new/2`.
- `named_table` true or false.
- `keypos` The position of the tuples which are the key position. The default is 1.

`info/1` returns `undefined` if the table does not exist.

`info(Tab, Item)`

Same as above, but only for the information that is associated with `Item`.

Except for the items mentioned above, these to items can be specified in calls to `info/2`:

- `fixed` Returns `true` if the table is fixed by any process, otherwise `false`. If the table identifier is no longer valid (deleted) the atom `undefined` is returned.
- `safe_fixed` If the table is 'fixed' using the `safe_fixtable` interface, the call returns a tuple: `{FixedNowTime,[{Pid,RefCount}]}`, where `FixedNowTime` is the time when the table was fixed by the first process (which may not be one of the processes fixing it now), `Pid` is a process 'fixing' the table right now and `RefCount` is the reference counter for 'fixes' done by that process. There may be any number of processes in the list.
  In *all* other cases, the atom `false` is returned.
  One can use this to write a monitor for 'fixed' tables if desired.

`tab2file(Tab, Filename)`

Dumps a table in the Erlang external term format to the file called `Filename`. Returns `ok`, or `{error, Reason}`. The function may crash if bad arguments are specified. The implementation of this function is not efficient.

`file2tab(Filename)`

Reads a file produced by the `tab2file/2` function and returns `{ok, Tab}` if the operation is successful, or `{error, Reason}` if it fails.

The error `{error, nofile}` is returned whenever the file cannot be read. This will be changed in future releases so that `{error, nofile}` is only returned when the file really does not exist, otherwise another error code will be returned. For applications that want to difference between errors, using the routines in the `file` module to detect if the file is nonexistent or inaccessible is to be preferred until this interface is changed.

`tab2list(Tab)`

Returns a list of all objects in the table.

`i()`

Displays a list of all local `ets` tables on the tty.

`i(Item)`

Browses an `ets` table on the tty. The `Item` argument is the identifier displayed in the left most field by the `i()` function.

filename (Module)

The module `filename` provides a number of useful functions for analyzing and manipulating file names. These functions are designed so that the Erlang code can work on many different platforms with different formats for file names. With file name is meant all strings that can be used to denote a file. They can be short relative names like `foo.erl`, very long absolute name which include a drive designator and directory names like `D:\usr/local\bin\erl/lib\tools\foo.erl`, or any variations in between.

In Windows, all functions return file names with forward slashes only, even if the arguments contain back slashes. Use the `join/1` function to normalize a file name by removing redundant directory separators.

Exports

`absname(Filename) -> Absname`

Types:

- Filename = string() | [string()] | atom()
- Absname = string()

Converts a relative `Filename` and returns an absolute name. No attempt is made to create the shortest absolute name, because this can give incorrect results on file systems which allow links.

Examples include:

```
Assume (for UNIX) current directory "/usr/local"
Assume (for WIN32) current directory "D:/usr/local"

(for UNIX):  absname("foo") -> "/usr/local/foo"
(for WIN32): absname("foo") -> "D:/usr/local/foo"
(for UNIX):  absname("../x") -> "/usr/local/../x"
(for WIN32): absname("../x") -> "D:/usr/local/../x"
(for UNIX):  absname("/") -> "/"
(for WIN32): absname("/") -> "D:/"
```

`absname(Filename, Directory) -> Absname`

Types:

- Filename = string() | [string()] | atom()
- Directory = string()
- Absname = string()

This function works like `absname/1`, except that the directory to which the file name should be made relative is given explicitly in the `Directory` argument.

`basename(Filename)`

Types:

- Filename = string() |[string()] | atom()

Returns the part of the `Filename` after the last directory separator, or the `Filename` itself if it has no separators.

Examples include:

```
basename("foo") -> "foo"
basename("/usr/foo") -> "foo"
basename("/") -> []
```

`basename(Filename,Ext) -> string()`

Types:

- Filename = Ext = string() | [string()] | atom()

Returns the last component of `Filename` with the extension `Ext` stripped. Use this function if you want to to remove an extension which might, or might not, be there. Use `rootname(basename(Filename))` if you want to remove an extension that exists, but you are not sure which one it is.

Examples include:

```
basename("~/src/kalle.erl", ".erl") -> "kalle"
basename("~/src/kalle.beam", ".erl") -> "kalle.beam"
basename("~/src/kalle.old.erl", ".erl") -> "kalle.old"
rootname(basename("~/src/kalle.erl")) -> "kalle"
rootname(basename("~/src/kalle.beam")) -> "kalle"
```

`dirname(Filename) -> string()`

Types:

- Filename = string() | [string()] | atom()

Returns the directory part of `Filename`.

Examples include:

```
dirname("/usr/src/kalle.erl") -> "/usr/src"
dirname("kalle.erl") -> "."
On Win32:
filename:dirname("\\usr\\src/kalle.erl") -> "/usr/src"
```

`extension(Filename) -> string() | []`

Types:

- Filename = string() | [string()] | atom()

Given a file name string `Filename`, this function returns the file extension including the period. Returns an empty list if there is no extension.

Examples include:

```
extension("foo.erl") -> ".erl"
extension("beam.src/kalle") -> []
```

`join(Components) -> string()`

Types:

- Components = [string()]

Joins a list of file name `Components` with directory separators. If one of the elements in the `Components` list includes an absolute path, for example "/xxx", the preceding elements, if any, are removed from the result.

The result of the `join` function is "normalized":

- There are no redundant directory separators.
- In Windows, all directory separators are forward slashes and the drive letter is in lower case.

Examples include:

```
join("/usr/local", "bin") -> "/usr/local/bin"
join(["/usr", "local", "bin"]) -> "/usr/local/bin"
join(["a/b///c/"] -> "a/b/c"
join(["B:a\\b///c/"] -> "b:a/b/c"  % On Windows only
```

`join(Name1, Name2) -> string()`

Types:

- Name1 = Name2 = string()

Joins two file name components with directory separators. Equivalent to `join([Name1,Name2])`.

`nativename(Path) -> string()`

Types:

- Path = string()

Converts a filename in `Path` to a form accepted by the command shell and native applications on the current platform. On Windows, forward slashes will be converted to backward slashes. On all platforms, the name will be normalized as done by `join/1`.

Example:

```
(on UNIX)  filename:nativename("/usr/local/bin/") -> "/usr/local/bin"
(on Win32) filename:nativename("/usr/local/bin/") -> "\\usr\\local\\bin"
```

`pathtype(Path) -> absolute | relative | volumerelative`

Returns one of `absolute`, `relative`, or `volumerelative`.

`absolute` The path name refers to a specific file on a specific volume.

Examples include:

```
on Unix
/usr/local/bin/
on Windows
D:/usr/local/bin
```

`relative` The path name is relative to the current working directory on the current volume.

Example:

```
foo/bar, ../src
```

`volumerelative` The path name is relative to the current working directory on a specified volume, or it is a specific file on the current working volume.

Examples include:

```
In Windows
D:bar.erl, /bar/foo.erl
/temp
```

`rootname(Filename) -> string()`
`rootname(Filename, Ext) -> string()`

Types:

- Filename = Ext = string() | [string()] | atom()

`rootname/1` returns all characters in `Filename`, except the extension.

`rootname/2` works as `rootname/1`, except that the extension is removed only if it is `Ext`.

Examples include:

```
rootname("/beam.src/kalle") -> "/beam.src/kalle"
rootname("/beam.src/foo.erl") -> "/beam.src/foo"
rootname("/beam.src/foo.erl",".erl") -> "/beam.src/foo"
rootname("/beam.src/foo.beam",".erl") -> "/beam.src/foo.beam"
```

`split(Filename) -> Components`

Types:

- Filename = string() | [string()] | atom()
- Components = [string()]

Returns a list whose elements are the path components of `Filename`.

Examples include:

```
split("/usr/local/bin") -> ["/", "usr", "local", "bin"]
split("foo/bar") -> ["foo", "bar"]
split("a:\\msdev\\include") -> ["a:/", "msdev", "include"]
```

`find_src(Module) -> {SourceFile, Options}`
`find_src(Module, Rules) -> {SourceFile, Options}`

Types:

- Module = atom() | string()
- SourceFile = string()
- Options = [CompilerOption]

- CompilerOption = {i, string()} | {outdir, string()} | {d, atom()}

Finds the source file name and compilation options for a compiled module. The result can be fed to `compile:file/2` in order to compile the file again.

The Module argument, which can be a string or an atom, specifies either the module name or the path to the source code, with or without the ".erl" extension. In either case, the module must be known by the code manager, i.e. `code:which/1` must succeed.

Rules describe how the source directory is found, when the object code directory is known. Each rule is of the form {`BinSuffix, SourceSuffix`} and is interpreted as follows: If the end of the directory name where the object is located matches `BinSuffix`, then the suffix of the directory name is replaced by `SourceSuffix`. If the source file is found in the resulting directory, then the function returns that location together with `Options`. Otherwise, the next rule is tried, and so on.

The function returns {`SourceFile, Options`}. `SourceFile` is the absolute path to the source file without the ".erl" extension. `Options` include the options which are necessary to compile the file with `compile:file/2`, but excludes options such as `report` or `verbose` which do not change the way code is generated. The paths in the {`outdir, Path`} and {`i, Path`} options are guaranteed to be absolute.

gen_event (Module)

`gen_event` provides a general framework for building application specific event handling routines. Event managers can be built for tasks like:

- error logging
- alarm handling
- call record logging
- debugging
- equipment management.

All event handlers are written as generic event managers and share a common set of interface functions. The generic parts of the event manager contains functions for debugging, handling the termination of the parent, and error handling.

The idea is that a server, the event manager, implements all server specific parts, while event handlers are added in order to handle specific events. Each event handler should be implemented in a module (called the callback module). Each callback module contains callback functions (e.g. `handle_event/2`) which are called whenever the event manager receives a corresponding message.

Event handlers can be written which act on *all* events, on *some* of the events, or on some particular combination of events. Event handlers can also be manipulated at runtime. In particular, an event handler can be:

- installed
- removed
- replaced by a different handler

We can even install several event handlers in the same event manager.

The relationship between the generic interface functions (and received messages) and the callback functions can be illustrated as follows:

```
Callback module  gen_event
----------------         ---------
gen_event:add_handler  ----->
Module:init/1            <-----

gen_event:notify        ----->
Module:handle_event/2   <-----

gen_event:call          ----->
Module:handle_call/2    <-----

gen_event:delete_handler ----->
Module:terminate/2       <-----
```

```
gen_event:stop          ----->
Module:terminate/2       <-----

gen_event:swap_handler         ----->
Mod1:terminate/2         <-----
Mod2:init/1              <-----

Module:handle_info/2    <-----         other message
                                received.
```

The event manager can be debugged using the `sys` module.

Exports

`start() -> ServerRet`
`start(Name) -> ServerRet`
`start_link() -> ServerRet`
`start_link(Name) -> ServerRet`

Types:

- Name = {local, atom()} | {global, atom()}
- ServerRet = {ok, Pid} | {error, Reason}
- Pid = pid()
- Reason = {already_started, Pid} | term()

This function starts an event manager. If the manager is started without `Name`, it can only be called by using the returned `Pid` identifier. If started with `Name`, the name is registered locally or globally.

An event manager started with `start/0` or `start/1` does not care about the parent. This means that the parent is not handled explicitly in the generic manager part. If started in this manner, these functions *must not* be used if the event manager is a worker in a supervision tree.

A manager started with `start_link/0` or `start_link/1` is initially linked to the caller - the parent - and it will terminate whenever the parent process terminates, with the same reason as the parent. An event manager always traps exit signals, so the `terminate/2` callback function is called for each added event handler in order to clean up before termination. If started in this manner, these functions *should* be used if the event manager is a worker in a supervision tree.

`stop(EventMgr) -> ok`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()

Terminates the event manager. The `terminate/2` callback function is called for each added event handler in order to clean up. The `Arg` argument of each `terminate/2` will have the value `stop`.

`notify(EventMgr, Event) -> ok`
`sync_notify(EventMgr, Event) -> ok`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Event = term()

Sends an event notification to the `EventMgr` event manager. The `Event` sent can be any Erlang term. However, the added event handlers must know about the term, and for this reason an event format must be specified for each event manager.

The event manager calls each associated `handle_event/2` callback function to inform each added event handler about the event.

The `notify/2` function is asynchronous, whereas `sync_notify/2` is synchronous in the sense that it returns when all handlers have handled the `Event`.

`add_handler(EventMgr, Handler, Args) -> ok | ErrorRet`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Handler = Module | {Module, Id}
- Module = atom()
- Id = term()
- Args = term()
- ErrorRet = term()

This function adds a new event handler to the `EventMgr` event manager. The callback module of the event handler is `Module` and the name of the handler is `Handler`. The `Id` term is used to identify a specific handler when installing several handlers which all use the same callback module. `Args` is supplied with the `Module:init(Args)` call in order to initialize the event handler. `ErrorRet` is any unexpected return value from the `init/1` function.

`add_sup_handler(EventMgr, Handler, Args) -> ok | ErrorRet`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Handler = Module | {Module, Id}
- Module = atom()

- Id = term()
- Args = term()
- ErrorRet = term()

Adds a new supervised event handler to the `EventMgr` event manager. The handler is added in the manner previously described for the `add_handler/3` function.

Whenever the process which evaluated this function terminates, the `Handler` is automatically deleted from the `EventMgr`. The `Module:terminate/2` function is called in order to clean up with `Arg` equal to {`stop, Reason`}. `Reason` is the termination reason of the process.

Whenever the `Handler` is deleted from the `EventMgr`, the process which evaluated this function receives the message {`gen_event_EXIT, Handler, Reason`}. `Reason` is one of the following:

- `normal`. The handler has been removed by the `delete_handler/3` function, or `remove_handler` has been returned by a callback function (see below).
- `shutdown`. The `EventMgr` process terminates, or the parent process of the handler terminates (the parent process could have sent an explicit `EXIT` signal to the `EventMgr` process and expects a message in response).
- {`swapped, NewHandler, NewParent`}. The handler has been replaced by `NewHandler` (see below).
- `Error`. The handler crashed due to `Error`. `Error` is any Erlang term (`term()`).

`delete_handler(EventMgr, Handler, Args) -> DelRet`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Handler = Module | {Module, Id}
- Module = atom()
- Id = term()
- Args = term()
- DelRet = term() | {error, module_not_found}

Removes the event handler `Handler` from the `EventMgr` event manager. `Args` is supplied with the `Module:terminate(Args, ...)` call in order to clean up the handler. Normally, it is preferable if `Args` is the atom `stop` as described for `stop/1`.

`DelRet` can be any Erlang term as returned from the `Module:terminate/2` function. This value can be used later on as a start argument (`Args = DelRet`) in order to restart (re-add) the same event handler with its old internal state. See also `swap_handler/3` below.

`swap_handler(EventMgr, OldHandler, NewHandler) -> SwRet`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()

- Pid = pid()
- OldHandler = {Handler1, Args1}
- NewHandler = {Handler2, Args2}
- Handler1 = Module1 | {Module1, Id1}
- Handler2 = Module2 | {Module2, Id2}
- Module1 = Module2 = atom()
- Id1 = Id2 = term()
- Args1 = Args2 = term()
- SwRet = ok | {error, SwErr}
- SwErr = term()

Removes the `Handler1` event handler and installs the new `Handler2` event handler. If appropriate, the new handler can inherit the internal state of the old handler.

`Module1:terminate(Args1,...)` is called to remove the old handler. The return value of the `terminate/2` function is passed to the new handler as `TermRet` below. The new handler is initialized by calling the `Module2:init({Args2,TermRet})` function in the new callback module. If an error occurs, the return value of the `init/1` function is returned as `SwErr`. To ignore the internal state of the old handler, the `TermRet` value should be ignored in the `init/1` function of the new handler.

If `Handler1` was added as a supervised handler, with the `add_sup_handler/3` function for example, the `Handler2` inherits the same parent. Thus, `Handler2` will be supervised by the same process as `Handler1`.

`swap_sup_handler(EventMgr, OldHandler, NewHandler) -> SwRet`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- OldHandler = {Handler1, Args1}
- NewHandler = {Handler2, Args2}
- Handler1 = Module1 | {Module1, Id1}
- Handler2 = Module2 | {Module2, Id2}
- Module1 = Module2 = atom()
- Id1 = Id2 = term()
- Args1 = Args2 = term()
- SwRet = ok | {error, SwErr}
- SwErr = term()

Removes the `Handler1` event handler and installs the new `Handler2` event handler in the same manner described for the `swap_handler/3` function above.

The `Handler2` event handler will be supervised by the process that evaluated this function, in the manner described for the `add_sup_handler/3` function above.

`call(EventMgr, Handler, Query) -> Ret`
`call(EventMgr, Handler, Query, Timeout) -> Ret`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid

- Name = atom()
- Node = atom()
- Pid = pid()
- Handler = Module | {Module, Id}
- Module = atom()
- Id = term()
- Query = term()
- Timeout = int() > 0 | infinity
- Ret = Reply | {error, ErrCall}
- Reply = term()
- ErrCall = bad_module | term()

Sends a request to the specified event handler `Handler` in the `EventMgr` event manager. `Query` can be any Erlang term, but it must be recognized by the event handler. To handle the request, the callback function `Module:handle_call/2` is called. `bad_module` is returned if the `Module` event handler does not exist. `Reply` is the returned `Reply` value of the callback function, while `ErrCall` is returned as an error descriptor if the callback module fails.

`Timeout` should be set to some reasonable value (in milliseconds). The special value `infinity` can be used if the user has no idea how long the request is supposed to take. If `Timeout` is not specified, the default value is `5000`.

If `Timeout` has an integer value and no response has been delivered within `Timeout` milliseconds, then the client will terminate with reason `{timeout, {gen_event, call, [EventMgr, Handler, Query, Timeout]}}`.

`which_handlers(EventMgr) -> [Handler]`

Types:

- EventMgr = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Handler = Module | {Module, Id}
- Module = atom()
- Id = term()

Asks the `EventMgr` event manager about active event handlers. This function returns a list of each added event handler.

Callback Functions

The following functions should be exported from a `gen_event` callback module.

Exports

`Module:init(Args) -> InitRes`

Types:

- Args = term()
- InitRes = {ok, State} | Other
- State = term()
- Other = term()

Whenever a new event handler is added to an event manager, the `init/1` function in the specified callback module is called in order to initialise the handler. If the initialization function succeeds, it is supposed to return the initialized internal `State` of the handler. The `State` is passed to all subsequent callback function calls to the handler.

The `Args` argument supplied to the `init/1` function is the same argument that is supplied to, for example, the `add_handler/3` function.

`Module:handle_event(Event, State) -> EventRet`

Types:

- Event = term()
- EventRet = {ok, State1} | {swap_handler, Args1, State1, Handler2, Args2} | remove_handler | Other
- Args1 = Args2 = term()
- State1 = State = term()
- Handler2 = Module | {Module, Id}
- Module = atom()
- Id = term()
- Other = term()

For each event handler, this function is called by the event manager whenever the event manager has received an event. `Event` is the value sent with the `gen_event:notify/2` function call. (Any other unmatched messages which are received by the event manager - such as `{'EXIT', Pid, Why}` - are processed using `handle_info/2`)

Normally, the event handler returns a new state with `{ok, State1}` after the event has been processed. The event handler can also remove itself or swap to another handler. If the handler is removed (returned `remove_handler`), the `terminate/2` callback function is called with `remove_handler` as the first argument. The swap procedure is the same as described for `swap_handler/3`.

If the `handle_event/2` function crashes, or returns `Other`, the `Module`:terminate/2 function is called in order to clean up (if possible) and the handler is removed from the event manager. The `Arg` argument of `Module`:terminate/2 is `{error,Reason}`, where `Reason` is `{'EXIT',Why}` if crashed, or `Other`.

`Module:handle_call(Query, State) -> CallRet`

Types:

- Query = term()
- CallRet = {ok, Reply, State1} | {swap_handler, Reply, Args1, State1, Handler2, Args2} | {remove_handler, Reply} | Other

- Reply = term()
- Args1 = Args2 = term()
- State1 = State = term()
- Handler2 = Module | {Module, Id}
- Module = atom()
- Id = term()
- Other = term()

Handles a request generated by a `call/3` function call. The request is dedicated to this handler. `Query` can be any Erlang term recognized by the event handler. The type of queries which are handled is a design issue. `Reply` is any Erlang term which represents the reply to the call. `Reply` is returned by the `call/3` function.

Normally, the event handler returns a new state with `{ok, Reply, State1}` after the call has been processed. The event handler can also decide to remove itself or to swap to another handler. If the handler should be removed (returned `{remove_handler, Reply}`), the `terminate/2` callback function is called with `remove_handler` as the first argument. The swap procedure is the same as described for `swap_handler/3`.

If the `handle_call/2` function crashes, or returns `Other`, the `Module`:terminate/2 function is called in order to clean up (if possible) and the handler is removed from the event manager. The `Arg` argument of `Module`:terminate/2 is `{error,Reason}`, where `Reason` is `{'EXIT',Why}` if crashed, or `Other`.

`Module:handle_info(Info, State) -> EventRet`

Types:

- Info = term()
- EventRet = {ok, State1} | {swap_handler, Args1, State1, Handler2, Args2} | remove_handler | Other
- Args1 = Args2 = term()
- State1 = State = term()
- Handler2 = Module | {Module, Id}
- Module = atom()
- Id = term()
- Other = term()

This callback function handles events other than `notify` and `call`, which are received by the event manager. Typical events, or messages, which are handled include:

`{'EXIT', Pid, Reason}` If the process traps exit signals, the corresponding messages are handled here.

`{nodedown, Node}` If another Erlang node is monitored, the corresponding `nodedown` message is handled here.

`Msg` All other messages, sent to the event manager using `EventMgr ! Msg`, are also handled here.

> **Note:**
> Communication with the event manager should always go through the above interface functions.

The `EventRet` value is the same as for `handle_event`.

`Module:terminate(Arg, State) -> TermRet`

Types:

- Arg = stop | remove_handler | {error, term()} | {stop, term()} | term()
- TermRet = term()

Cleans up the event handler before it is removed from the event manager. If `Arg` is `stop` or `remove_handler`, the event handler is supposed to be removed and no other handler is supposed to take over the internal state. In this case, `TermRet` is ignored.

If another handler is taking over the internal state of this handler, this should be marked with `Arg` as some other Erlang term, `swap` for example. In this case, the event handler should return the internal state `State`, or parts of the state, in a way that is recognized by the handler which is supposed to take over.

`Arg` is `{error, Error}` if a callback function has crashed or returned something inappropriate. `Error` is `{'EXIT', Why}` if it has crashed.

`Arg` is `{stop, Reason}` if the parent of a supervised event handler has terminated. `Reason` is the termination reason for the parent process.

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`

Types:

- OldVsn = undefined | term()
- State = term()
- Extra = term()
- NewState = term()

This function is called when a code change is performed, which implies that the internal data structures of the `Module` event handler has changed. This function is supposed to convert the old state to the new one. `OldVsn` is the *vsn* attribute of the old version of the module. If no such attribute was defined, the atom `undefined` is sent. `Extra` is an optional term, which is typically defined in the release upgrade script.

System events

The gen_event behaviour generates the following system events, which are handled by the `sys` module:

- `{in, Msg}` when a message is received.

See also

sys(3)

gen_fsm (Module)

This module provides a standard way of writing Finite State Machine (FSM) processes. All FSMs written as gen_fsms share a common set of interface functions. The generic parts of the FSM contains functions for debugging, for handling the termination of the parent process, and for presentation of illustrative error information if something goes wrong in the process.

The state of the FSM is defined by two parameters, the `StateName` and the `StateData`. For each `StateName`, there must be a corresponding function exported from the call-back module. When an event is received, and the current state of the FSM is `StateName`, `Module:StateName(Event, StateData)` is called. This function should return the next state, which is the next `StateName`.

It is also possible to define a function `Module:handle_event(Event, StateName, StateData)` to take care of events which should always be handled, regardless of their state. This function is called when `gen_fsm:send_all_state_event/2` is used to generate an event.

Events can be handled synchronously as well. This means that the caller waits for a reply to the event.

The relationship between the generic interface functions (and received messages) and the callback functions can be illustrated as follows:

```
Callback module                            gen_fsm
----------------                            -------
gen_fsm:start_link                 ----->   start a new fsm process
Module:init/1                     <-----
                                            looping

gen_fsm:send_event                ----->
Module:StateName/2                 <-----

gen_fsm:sync_send_event           ----->
Module:StateName/3                 <-----

gen_fsm:send_all_state_event      ----->
Module:handle_event/3              <-----

gen_fsm:sync_send_all_state_event ----->
Module:handle_sync_event/4         <-----


Module:handle_info/3               <-----     other message
                                             received.

Module:terminate/3                 <-----     clean up before
                                             termination.
```

> **Note:**
> Trapping of exits, if required, must be done explicitly.

An instance of the `gen_fsm` behaviour can be debugged by using the module `sys`.

Exports

```
start(Module, StartArgs, Options) -> StartRet
start_link(Module, StartArgs, Options) -> StartRet
start(Name, Module, StartArgs, Options) -> StartRet
start_link(Name, Module, StartArgs, Options) -> StartRet
```

Types:

- Name = {local, atom()} | {global, atom()}
- Module = atom()
- StartArgs = term()
- Options = [Opt]
- Opt = {debug, [Dbg]} | {timeout, Time}
- Dbg = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
- StartRet = {ok, Pid} | ignore | {error, Reason}
- Pid = pid()
- Reason = {already_started, Pid} | term()

Starts an FSM process. An anonymous process is started if `Name` is not specified. This process can only be called by using the returned `Pid` identifier.

A process which is started with `start` does not care about the parent, which means that the parent is not handled explicitly in the generic process part. If started in this manner, this function *must not* be used if the FSM is a worker in a supervision tree.

A process started with `start_link` is initially linked to the caller - the parent - and will terminate whenever the parent process terminates, and with the same reason as the parent. If started in this manner, this function *should* be used if the FSM is a worker in a supervision tree.

The function `Module:init(StartArgs)` is called (see below).

`Time` specifies how long time, in milliseconds, the server is allowed to initialize itself.

The debug options are described in `sys(3)`.

```
send_event(ProcessRef,Event) -> void()
```

Types:

- ProcessRef = Name | {Name, Node} | {global, Name} | pid()
- Name = atom()
- Node = atom()

- Event = term()

Sends an event asynchronously to the FSM process. In the callback module, the function `StateName/2` is called, where `StateName` is the name of the current state.

`send_all_state_event(ProcessRef,Event) -> void()`

Types:

- ProcessRef = Name | {Name, Node} | {global, Name} | pid()
- Name = atom()
- Node = atom()
- Event = term()

An event, which can be handled in all states, is sent asynchronously to the FSM process. In the callback module, `handle_event/3` is called.

`sync_send_event(ProcessRef,Event) -> Reply`
`sync_send_event(ProcessRef,Event, Timeout) -> Reply`

Types:

- ProcessRef = Name | {Name, Node} | {global, Name} | pid()
- Name = atom()
- Node = atom()
- Event = term()
- Timeout = int() > 0 | infinity
- Reply = term()

Sends an event synchronously to the FSM process and waits for the answer. In the callback module, the function `StateName/3` is called, where `StateName` is the name of the current state.

`Timeout` should be set to some reasonable value. The special value `infinity` can be used if the user has no idea how long the request is supposed to take. The default is 5000.

If `Timeout` has an integer value and if no response has been delivered within `Timeout` milliseconds, the client will terminate with reason `{timeout, {gen_fsm, sync_send_event, [ProcessRef, Event, Timeout]}}`.

`sync_send_all_state_event(ProcessRef,Event) -> Reply`
`sync_send_all_state_event(ProcessRef,Event,Timeout) -> Reply`

Types:

- ProcessRef = Name | {Name, Node} | {global, Name} | pid()
- Name = atom()
- Node = atom()
- Event = term()
- Timeout = int() > 0 | infinity
- Reply = term()

An event, which can be handled in all states, is sent synchronously to the FSM process. In the callback module, `handle_event/4` is called.

`Timeout` should be set to some reasonable value. The special value `infinity` can be used if the user has no idea how long the request is supposed to take. The default is 5000.

If `Timeout` has an integer value and no response has been delivered within `Timeout` milliseconds, the client will terminate with reason `{timeout, {gen_fsm, sync_send_all_state_event, [ProcessRef, Event, Timeout]}}`.

`reply(To, Reply) -> true`

Types:

- To = {pid(), Tag}
- Tag = term()
- Reply = term()

If a reply cannot be returned immediately - as the return value of `Module:StateName/3` or `Module:handle_sync_event/4` - this function can be used to make an explicit reply. `To` has the same value as the `From` argument in these functions.

Callback Functions

The following functions should be exported from a `gen_fsm` callback module.

Exports

`Module:init(StartArgs) -> Return`

Types:

- StartArgs = term()
- StateName = atom()
- StateData = term()
- Timeout = int() > 0 | infinity
- StopReason = term()
- Return = {ok, StateName, StateData} | {ok, StateName, StateData, Timeout} | ignore | {stop, StopReason}

This function initializes the FSM process and returns the initial state. The `Timeout` variable specifies that the process shall wait for `Timeout` milliseconds for the first message. If no message has arrived within the specified time, `Module:StateName(timeout, StateData)` is called.

The `StartArgs` argument supplied to the `init/1` function is the same as the argument supplied to the `gen_fsm:start` functions.

If the process should trap exits, this has to be explicitly expressed here with `process_flag(trap_exit, true)`.

The representation of the FSM `StateData` is an implementation specific detail which has to be decided by the designer of the FSM. It can be any Erlang term. `StateData` will be visible as an argument to all callback functions. To change something in `StateData`, a new value is returned from the callback function using the terms described below.

If the initializing procedure fails, the reason is supplied as `StopReason` with the `{stop, StopReason}` return value.

This function can return `ignore` in order to inform the parent, especially if it is a supervisor, that the FSM, as an example, has not started in accordance with the configuration data.

`Module:StateName(Event, StateData) -> Return`

Types:

- Event = term()
- StateData = term()
- Return = {next_state, NextStateName, NextStateData} | {next_state, NextStateName, NextStateData, Timeout} | {stop, Reason, NewStateData}
- NextStateName = atom()
- NextStateData = term()
- Reason = normal | shutdown | term()

Handles events in the state `StateName`. The `Timeout` variable is as in `Module:init/1` above.

Whenever the function `gen_fsm:send_event` is called, this function is called to handle the event. If the FSM times out, this function is also called with `Event = timeout`.

`Event` is the same term as supplied in the above client call.

If the FSM decides to terminate, this function should return `{stop, Reason, NewStateData}`, and the function `Module:terminate(Reason, StateName, NewStateData)` is called. If `Reason` is something other than `normal` or `shutdown`, the FSM is assumed to have terminated with a runtime failure. In this case, a lot of information about the failure is reported. The atom `normal` causes a normal termination while `shutdown` causes an abnormal, but faultless, termination of the process.

`Module:StateName(Event, From, StateData) -> Return`

Types:

- Event = term()
- From = {pid(), Tag}
- StateData = term()
- Return = {next_state, NextStateName, NextStateData} | {next_state, NextStateName, NextStateData, Timeout} | {reply, Reply, NextStateName, NextStateData} | {reply, Reply, NextStateName, NextStateData, Timeout} | {stop, Reason, NewStateData} | {stop, Reason, Reply, NewStateData}
- NextStateName = atom()
- NextStateData = term()
- Reply = term()
- Reason = normal | shutdown | term()

Handles synchronous events in the state `StateName`. The `Timeout` variable is as in `Module:init/1` above.

Whenever the function `gen_fsm:sync_send_event/2,3` is called, this function is called to handle the event.

`Event` is the same as the term supplied with the above client call.

The FSM decides if a reply is sent to the caller directly (`{reply, ...}`), indirectly (`{next_state, ...}`), or if the FSM has to terminate (`{stop, ...}`) as a result of the request. If `{next_state, ...}` is returned, a reply can be sent to the caller using the `reply/2` function.

If the FSM decides to terminate, this function returns `{stop, Reason, NewStateData}` or `{stop, Reason, Reply, NewStateData}`, and the function `Module:terminate(Reason, StateName, NewStateData)` is called. If `Reason` is something other than `normal` or `shutdown`, the FSM is assumed to have terminated with a runtime failure. In this case, a lot of information about the failure is reported. The atom `normal` causes a normal termination while `shutdown` causes an abnormal, but faultless, termination of the process.

`Module:handle_event(Event, StateName, StateData) -> Return`

Types:

- Event = term()
- StateName = atom()
- StateData = term()

Handles events generated with the function `gen_fsm:send_all_state_event/2`.

The `Return` value is the same as for `Module:StateName/2`.

`Module:handle_sync_event(Event, From, StateName, StateData) -> Return`

Types:

- Event = term()
- From = {pid(), Tag}
- StateName = atom()
- StateData = term()

Handles events generated with the function `gen_fsm:sync_send_all_state_event/2,3`.

The `Return` value is the same as for `Module:StateName/3`.

`Module:handle_info(Info, StateName, StateData) -> Return`

Types:

- Info = term()
- StateName = atom()
- StateData = term()

This function receives all messages sent to this process which are not generated by `gen_fsm:send_event/2`, `gen_fsm:send_all_state_event/2`, `gen_fsm:sync_send_event/2,3`, or `gen_fsm:sync_send_all_state_event/2,3`. Typical messages handled here include:

`{'EXIT', Pid, Reason}` If the process traps exit signals, the corresponding messages are handled here.

`{nodedown, Node}` If another Erlang node is monitored, the corresponding `nodedown` message is handled here.

`Msg` All other messages sent to the process using `Fsm ! Msg` are also handled here.

> **Note:**
> Communication with the FSM should always go through the interface functions described above.

The `Return` value is the same as for `Module:StateName/2`.

`Module:terminate(Reason, StateName, StateData) -> void()`

Types:

- Reason = term()
- StateName = atom()
- StateData = term()

This callback function is called whenever the FSM is about to terminate. Either one of the above callback functions have returned `{stop, StopReason, ...}`, in which case `Reason` is equal to `StopReason`; or some other fault has been caught. `Reason` is any term which describes the termination reason. If the FSM traps exits, the `terminate` function is called if the FSM's parent (normally a supervisor) dies or orders the FSM to die. If the FSM does not trap exits, it dies immediately if the parent dies.

With this function, the FSM can clean up before the process terminates. It can, for example, de-allocate external resources.

The termination reason cannot be changed here. The FSM will terminate due to`Reason` regardless of what was returned from this function.

`Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NewState, NewStateData}`

Types:

- OldVsn = undefined | term()
- StateName = atom()
- StateData = term()
- Extra = term()
- NewStateName = atom()
- NewStateData = term()

This function is called when a code change is performed, which implies that the internal data structures of the FSM have changed. The function is supposed to convert the old state to the new one. `OldVsn` is the *vsn* attribute of the old version of the module. If no such attribute was defined, the atom `undefined` is sent. `Extra` is an optional term, typically defined in the release upgrade script.

System events

The `gen_fsm` behaviour generates the following system events, handled by the `sys` module:

- `{in, Msg}` when a message is received.
- `{out, Msg, To, StateName}` when a message is sent.
- `return` when an event handling callback function returns.

See Also

sys(3)

gen_server (Module)

This module provides a standard way of writing Client-Server applications. All servers written as generic servers share a common set of interface functions. The generic parts of the server contain functions for debugging, handling the termination of the parent, and presentation of illustrative error information if something goes wrong with the server.

The idea is that the implementation specific parts of a client-server is in one module, called the callback module. The callback module contains the client interface functions which use the server access functions described below. The callback module also contains the server callback functions, for example `handle_call/3`. Whenever the generic part of the server receives a message - sent through a server access function, for example - the corresponding callback function is called.

The relationship between the generic interface functions (and received messages) and the callback functions can be illustrated as follows:

```
Callback module                gen_server
----------------                ----------
gen_server:start      ----->    start a new server
Module:init/1         <-----
                                looping

gen_server:call       ----->
Module:handle_call/3  <-----

gen_server:cast       ----->
Module:handle_cast/2  <-----

gen_server:multi_call ----->
Module:handle_call/3  <-----

gen_server:abcast     ----->
Module:handle_cast/2  <-----

Module:handle_info/2  <-----  other message
                              received.

Module:terminate/2    <-----  clean up before
                              termination.
```

If the server wants to trap exit signals, this must be explicitly initiated in the callback module.

An instance of the `gen_server` behaviour can be debugged using the module `sys`.

Exports

```
start(Module, Args, Options) -> ServerRet
start(ServerName, Module, Args, Options) -> ServerRet
start_link(Module, Args, Options) -> ServerRet
start_link(ServerName, Module, Args, Options) -> ServerRet
```

Types:

- Module = atom()
- ServerName = {local, atom()} | {global, atom()}
- Args = term()
- Options = [Opt]
- Opt = {debug, [Dbg]} | {timeout, Time}
- Dbg = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
- ServerRet = {ok, Pid} | ignore | {error, Reason}
- Pid = pid()
- Reason = {already_started, Pid} | term()

Starts a new server. If the server is started without `ServerName`, it can only be called using the returned `Pid` identifier. If started with `ServerName`, the name is registered locally or globally.

`Module` is the name of the callback module.

A server started with `start/3` or `start/4` does not care about the parent, which means that the parent is not handled explicitly in the generic process part. If started in this manner, these functions *must not* be used if the server is a worker in a supervision tree.

A server started with `start_link/3` or `start_link/4` is initially linked to the caller, the parent, and it will terminate whenever the parent process terminates, and with the same reason as the parent. If the server traps exits, the `terminate/2` callback function is called in order to clean up before the termination. If started in this manner, these functions *should* be used if the server is a worker in a supervision tree.

`Time` specifies how long time, in milliseconds, the server is allowed to spend initializing.

The function `Module:init(Args)` is called in the new process in order to initialize the server (see below).

Refer to the `sys` module for more information about the `Dbg` options.

```
call(ServerRef, Request) -> Reply
call(ServerRef, Request, Timeout) -> Reply
```

Types:

- ServerRef = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Request = term()
- Timeout = int() > 0 | infinity
- Reply = term()

A request is sent to the `ServerRef` server. The request can be any term, but the term must be recognized by the server. The request is handled by the server (in the `Module:handle_call/2` function) and the client is suspended while waiting for the response. `Timeout` should be set to some reasonable value in milliseconds. The special value `infinity` can be used if the user has no idea how long the request is supposed to take. The default value is `5000` if `Timeout` is not specified.

If `Timeout` has an integer value and no response has been delivered within `Timeout` milliseconds, then the client terminates with reason `{timeout, {gen_server, call, [ServerRef, Request, Timeout]}}`.

```
cast(ServerRef, Request) -> ok
```

Types:

- ServerRef = Name | {Name, Node} | {global, Name} | Pid
- Name = atom()
- Node = atom()
- Request = term()

A request is sent to the server. As no response will be delivered, the client making the cast is not suspended until the request has been handled by the server. This function returns `ok` immediately and ignores non-existing servers.

```
multi_call(DistRef, Request) -> DistRep
multi_call(Nodes, DistRef, Request) -> DistRep
```

Types:

- Nodes = [Node]
- Node = atom()
- DistRef = atom()
- DistRep = {[{Node,Reply}],[Node]}
- Request = term()
- Reply = term()

Sends a request to the locally registered server `DistRef` at every known node (or `Nodes`). This function returns a list of replies which are tagged with the corresponding node name, and a list of bad nodes. `Reply` is the value returned by a server. A node is marked bad if the server at a specific node, or the node itself, does not exist.

The request is sent to the `DistRef` server at all nodes before the replies are collected. This ensures that the request is handled in parallel on all nodes. If no reply is received during the collection of replies, the existence of a server at a specific node is checked. This function hangs if the server was started after the request was sent, but before the existence check was performed

```
abcast(DistRef, Request) -> abcast
abcast(Nodes, DistRef, Request) -> abcast
```

Types:

- Nodes = [Node]
- Node = atom()
- DistRef = atom()
- Request = term()

Broadcasts the request asynchronously to the locally registered server `DistRef` on every known node (or `Nodes`). This function returns immediately and ignores non-existing servers or nodes.

`reply(To, Reply) -> true`

Types:

- To = {pid(), Tag}
- Tag = term()

This function can be used by a server to make an explicit reply, if a reply cannot be returned immediately as the return value of `Module:handle_call/3`. `To` has the same value as the `From` argument in `Module:handle_call/3`.

Callback Functions

The following functions should be exported from a `gen_server` callback module.

Exports

`Module:init(Args) -> {ok, State} | {ok, State, Timeout} | ignore | {stop, StopReason}`

Types:

- Args = term()
- State = term()
- Timeout = int() >= 0 | infinity
- StopReason = term()

Whenever a new server is started, `init/1` is the first function called in the specified callback module. To ensure a synchronized start-up procedure, the `gen_server:start` function will not return before the `init/1` function has returned.

The `Args` argument supplied to the `init/1` function is the same as the `Args` parameter supplied to the `gen_server:start` functions.

The purpose of the `init/1` function is to initialize the server and the internal state of the server. A server which holds an external resource typically opens the associated port and keeps the port identity in the internal state.

If the server wants to trap exits, this has to be expressed explicitly in the init function with `process_flag(trap_exit, true)`.

The representation of the server `State` is an implementation specific detail which must be decided by the designer of the server. `State` will be visible as an argument to all callback functions. To change something in `State`, a new value is returned from the callback function using the return values (terms) described below.

If the initializing procedure fails, the reason is supplied as `StopReason` with the {`stop`, `StopReason`} return value.

After the server has been successfully initialized, the generic part of the server enters the main loop and waits for requests. A `Timeout` time can be specified if the server is only allowed to wait for a certain time for the next event. If the timeout time elapses, the special `timeout` message should be handled in the `Module:handle_info/2` callback function. `Timeout` is specified in milliseconds.

This function can return `ignore` in order to inform the parent, especially if it is a supervisor, that the server, as an example, did not start in accordance with the configuration data.

`Module:handle_call(Request, From, State) -> CallReply`

Types:

- Request = term()
- From = {pid(), Tag}
- Tag = term()
- CallReply = {reply, Reply, State} | {reply, Reply, State, Timeout} | {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, Reply, State} | {stop, StopReason, State}
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

Whenever a client function has called one of the interface functions `gen_server:call` or `gen_server:multi_call`, the server handles the request in this callback function.

`Request` is the same as the term supplied with the above client call. The server decides if the client should be sent a reply directly (`{reply, ...}`), indirectly (`{noreply, ...}`), or if the server has to terminate (`{stop, ...}`) as a result of the request. If `{noreply, ...}` is returned, a reply is sent to the client using the `reply/2` function.

If `StopReason` is something other than `normal` or `shutdown`, the server is assumed to have terminated with a runtime error. In this case, a lot of information is reported about the failure. The atom `normal` causes a normal termination of the server, while `shutdown` causes an abnormal, but faultless, termination.

If the server decided to terminate `{stop, StopReason [, ...]}`, the `Module:terminate/2` function is called. All code which handles the clean up before the server terminates should be located in the `terminate` function. The server will terminate due to `StopReason`.

As described above (see `init/1`), a timeout can be specified to take some specific action if no more requests are received within `Timeout` milliseconds.

`Module:handle_cast(Request, State) -> Return`

Types:

- Request = term()
- State = term()
- Return = {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, State}
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

Whenever a client function has called one of the interface functions `gen_server:cast` or `gen_server:abcast`, the server handles the request in this callback function. No reply will ever be sent to the client, but the server can decide to terminate. `StopReason` is as described for `handle_call/3`.

`Module:handle_info(Info, State) -> Return`

Types:

- Info = term()
- State = term()
- Return = {noreply, State} | {noreply, State, Timeout} | {stop, StopReason, State}
- Timeout = int() >= 0 | infinity
- StopReason = normal | shutdown | term()

This callback function handles received messages other than `call` and `cast`. Typical messages which are handled by this function include:

`{'EXIT', Pid, Reason}` If the process traps exit signals, the corresponding messages are handled here.

`{nodedown, Node}` If another Erlang node is monitored, the corresponding `nodedown` message is handled here.

`timeout` If `Timeout` milliseconds has elapsed since the last handled event, this message should be handled.

`Msg` All other messages which are sent to the server using `Server ! Msg` are also handled here.

> **Note:**
> Communication with the server should always go through the interface functions described above.

The `Return` value is the same as for `handle_cast/2`. `StopReason` is as described for `handle_call/3`.

`Module:terminate(Reason, State) -> ok`

Types:

- Reason = term()
- State = term()

This callback function is called whenever the server is about to terminate. Either one of the above callback functions have returned `{stop, StopReason, ...}`, in which case `Reason` is equal to `StopReason`; or some other fault has been caught. `Reason` is any term which describes the termination reason. If the server traps exits, the `terminate` function is called if the server's parent (normally a supervisor) dies or orders the server to die. If the server does not trap exits, it dies immediately if the parent dies.

With this function, the server can clean up before the process terminates. It can, for example, de-allocate external resources.

The termination reason cannot be changed here. The server will terminate due to`Reason` regardless of what was returned from this function.

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`

Types:

- OldVsn = undefined | term()
- State = term()
- Extra = term()
- NewState = term()

This function is called when a code change is performed, which implies that the internal data structures of the server has changed. This function is supposed to convert the old state to the new one. `OldVsn` is the *vsn* attribute of the old version of the module. If no such attribute was defined, the atom `undefined` is sent. `Extra` is an optional term which is typically defined in the release upgrade script.

System Events

The `gen_server` behaviour generates the following system events, handled by the `sys` module:

- `{in, Msg}` when a message is received.
- `{out, Msg, To}` when a message is sent.
- `{noreply, State}` when no reply is delivered.

Example

The following example implements a simple queue server. The server has four interface functions:

- `start/0` which starts the queue server.
- `stop/0` which stops the queue server.
- `in/1` which inserts an item last in the queue.
- `out/0` which removes the oldest item from the queue.

The queue server is not linked to the parent process and the server does not handle the termination of the parent process explicitly.

```
-module(queue_serv).
-behaviour(gen_server).

%% External exports
-export([start/0, in/1, out/0, stop/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2]).
```

```
start() ->  gen_server:start({local, queue_serv},
                             queue_serv, [], []).

in(Item) -> gen_server:call(queue_serv, {in, Item}).

out() -> gen_server:call(queue_serv, out).

stop() -> gen_server:call(queue_serv, stop).

%% Callback functions.
init([]) ->
    {ok, {[],[]}}.

handle_call({in, X}, _From, {In, Out}) ->
    {reply, ok, {[X|In], Out}};
handle_call(out, _From, Queue) ->
    {Reply, NewQueue} = out(Queue),
    {reply, Reply, NewQueue};
handle_call(stop, _From, Queue) ->
    {stop, normal, ok, Queue}.

handle_cast(_, State) ->
    {noreply, State}.

handle_info(_, State) ->
    {noreply, State}.

terminate(Reason, State) ->
    ok.

%% Internal functions
out({In, [H|Out]}) ->
    {{value, H}, {In, Out}};
out({[], []}) ->
    {empty, {[],[]}};
out({In, _}) ->
    out({[], lists:reverse(In)}).
```

See Also

sys(3)

io (Module)

This module provides an interface to standard Erlang IO servers. The output functions all return `ok` if they are successful, or exit if they are not. In the following description, a parameter within square brackets means that that parameter is optional. `[IoDevice,]` is such an example. If included, it must be the Pid of a process which handles the IO protocols. This is often the `IoDevice` returned by `file:open/2` (see `file`). For a description of the I/O protocols refer to Armstrong, Virding and Williams, 'Concurrent Programming in Erlang', Chapter 13.

Exports

`put_chars([IoDevice,] Chars)`

Writes the characters `Chars` to the standard output (`IoDevice`). `Chars` is a list of characters. The list is not necessarily flat.

`nl([IoDevice])`

Writes new line to the standard output (`IoDevice`).

`get_chars([IoDevice,] Prompt, Count)`

Gets `Count` characters from standard input (`IoDevice`), prompting it with `Prompt`. It returns:

`ListOfChars` Returns the input characters, if they are less than `Count`.

`eof` End of file was encountered.

`get_line([IoDevice,] Prompt)`

Gets a line from the standard input (`IoDevice`), prompting it with `Prompt`. It returns:

`ListOfChars` The characters in the line terminated by a LF unless the line read was the last line of the file and was not terminated by LF.

`eof` End of file was encountered.

`write([IoDevice,] Term)`

Writes the term `Term` to the standard output (`IoDevice`).

`read([IoDevice,] Prompt)`

Reads a term from the standard input (`IoDevice`), prompting it with `Prompt`. It returns:

{`ok, Term`} The parsing was successful.

{`error, ErrorInfo`} The parsing failed.

`eof` End of file was encountered.

`fwrite(Format)`
`format(Format)`

Equivalent to `fwrite(Format, []).`

`fwrite([IoDevice,] Format, Arguments)`
`format([IoDevice,] Format, Arguments)`

Writes the list of items in `Arguments` on the standard output (`IoDevice`) in accordance with `Format`. `Format` is a list of plain characters which are copied to the output device, and control sequences which cause the arguments to be printed. If `Format` is an atom, it is first converted to a list with the aid of `atom_to_list/1`. `Arguments` is the list of items to be printed.

```
> io:fwrite("Hello world!~n", []).
Hello world
ok
```

The general format of a control sequence is `~F.P.PadC`. The character `C` determines the type of control sequence to be used, `F` and `P` are optional numeric arguments. If `F`, `P`, or `Pad` is `*`, the next argument in `Arguments` is used as the numeric value of `F` or `P`.

`F` is the `field width` of the printed argument. A negative value means that the argument will be left justified within the field, otherwise it will be right justified. If no field width is specified, the required print width will be used. If the field width specified is too small, then the whole field will be filled with `*` characters.

`P` is the `precision` of the printed argument. A default value is used if no precision is specified. The interpretation of precision depends on the control sequences. Unless otherwise specified, the argument `within` is used to determine print width.

`Pad` is the padding character. This is the character used to pad the printed representation of the argument so that it conforms to the specified field width and precision. Only one padding character can be specified and, whenever applicable, it is used for both the field width and precision. The default padding character is `' '` (space).

The following control sequences are available:

`~` The character `~` is written.

`c` The argument is a number that will be interpreted as an ASCII code. The precision is the number of times the character is printed and it defaults to the field width, which in turn defaults to one. The following example illustrates:

```
> io:fwrite("|~10.5c|~-10.5c|~5c|~n", [$a, $b, $c]).
|     aaaaa|aaaaa     |ccccc|
ok
```

`f` The argument is a float which is written as `[-]ddd.ddd`, where the precision is the number of digits after the decimal point. The default precision is 6.

`e` The argument is a float which is written as `[-]d.ddde+-ddd`, where the precision is the number of digits written. The default precision is 6.

`g` The argument is a float which is written as `f`, if it is > 0.1, and < 10^4. Otherwise, it is written as `e`. The precision is the number of significant digits. It defaults to 6. There must always be a sufficient number of digits for printing a correct floating point representation of the argument.

`s` Prints the argument with the `string` syntax. The argument is a list of character codes (possibly not a flat list), or an atom. The characters are printed without quotes. In this format, the printed argument is truncated to the given precision and field width.

This format can be used for printing any object and truncating the output so it fits a specified field:

```
> io:fwrite("|~10w|~n", [{hey, hey, hey}]).
|**********|
ok
> io:fwrite("|~10s|~n", [io_lib:write({hey, hey, hey})]).
|{hey, hey, h|
ok
```

`w` Writes data with the standard syntax. This is used to output Erlang terms. Atoms are printed within quotes if they contain embedded non-printable characters, and floats are printed in the default g format.

`p` Writes the data with standard syntax in the same way as `~w`, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings. For example:

```
> T = [{attributes,[[{id,age,1.50000},{mode,explicit},
       {typename,"INTEGER"}],
       [{id,cho},{mode,explicit},{typename,'Cho'}]]},
       {typename,'Person'},{tag,{'PRIVATE',3}},
       {mode,implicit}].
...
> io:fwrite("~w~n", [T]).
[{attributes,[[{id,age,1.50000},{mode,explicit},{typename,
[73,78,84,69,71,69,82]}],[{id,cho},{mode,explicit},{typena
me,'Cho'}]]},{typename,'Person'},{tag,{'PRIVATE',3}},{mode
,implicit}]
ok
> io:fwrite("~p~n", [T]).
[{attributes,[[{id,age,1.50000},
               {mode,explicit},
               {typename,"INTEGER"}],
              [{id,cho},{mode,explicit},{typename,'Cho'}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

The field width specifies the maximum line length. It defaults to 80. The precision specifies the initial indentation of the term. It defaults to the number of characters printed on this line in the same call to `io:fwrite` or `io:format`. For example, using `T` above:

```
> io:fwrite("Here T = ~p~n", [T]).
Here T = [{attributes,[[{id,age,1.50000},
                        {mode,explicit},
                        {typename,"INTEGER"}],
                       [{id,cho},{mode,explicit},
                        {typename,'Cho'}]]},
          {typename,'Person'},
          {tag,{'PRIVATE',3}},
          {mode,implicit}]
ok
```

`W` Writes data in the same way as `~w`, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with `...`. For example, using `T` above:

```
> io:fwrite("~W~n", [T,9]).
[{attributes,[[{id,age,1.50000},{mode,explicit},{typename|
...},[{id,cho},{mode|...},{...}]]},{typename,'Person'},{t
ag,{'PRIVATE',3}},{mode,implicit}]
ok
```

If the maximum depth has been reached, then it is impossible to read in the resultant output. Also, the `|...` form in a tuple denotes that there are more elements in the tuple but these are below the print depth.

`P` Writes data in the same way as `~p`, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with `...`. For example:

```
> io:fwrite("~P~n", [T,9]).
[{attributes,[[{id,age,1.50000},{mode,explicit},
               {typename|...}],
              [{id,cho},{mode|...},{...}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

`n` Writes a new line.

`i` Ignores the next term.

Returns:

`ok` The formatting succeeded.

If an error occurs, there is no output. For example:

```
> io:fwrite("~s ~w ~i ~w ~c ~n",['abc def', 'abc def',
                                 {foo, 1},{foo, 1}, 65]).
abc def 'abc def'  {foo, 1} A
ok
> io:fwrite("~s", [65]).
** exited: {badarg,[{io,format,[<0.21.0>,"~s","A"]},
                    {erl_eval,expr,3},
                    {erl_eval,exprs,4},
                    {shell,eval_loop,2}]} **
```

In this example, an attempt was made to output the single character '65' with the aid of the string formatting directive "~s".

The two functions `fwrite` and `format` are identical. The old name `format` has been retained for backwards compatibility, while the new name `fwrite` has been added as a logical complement to `fread`.

`fread([IoDevice,] Prompt, Format)`

Reads characters from the standard input (`IoDevice`), prompting it with `Prompt`. Interprets the characters in accordance with `Format`. `Format` is a list of control sequences which directs the interpretation of the input.

`Format` may contain:

- White space characters (SPACE, TAB and NEWLINE) which cause input to be read to the next non-white space character.
- Ordinary characters which must match the next input character.
- Control sequences, which have the general format `~*FC`. The character `*` is an optional return suppression character. It provides a method to specify a field which is to be omitted. `F` is the `field width` of the input field and `C` determines the type of control sequence.

  Unless otherwise specified, leading white-space is ignored for all control sequences. An input field cannot be more than one line wide. The following control sequences are available:

  `~` A single `~` is expected in the input.

  `d` A decimal integer is expected.

  `f` A floating point number is expected. It must follow the Erlang floating point number syntax.

  `s` A string of non-white-space characters is read. If a field width has been specified, this number of characters are read and all trailing white-space characters are stripped. An Erlang string (list of characters) is returned.

  `a` Similar to `s`, but the resulting string is converted into an atom.

  `c` The number of characters equal to the field width are read (default is 1) and returned as an Erlang string. However, leading and trailing white-space characters are not omitted as they are with `s`. All characters are returned.

  `l` Returns the number of characters which have been scanned up to that point, including white-space characters.

  It returns:

  `{ok, InputList}` The read was successful and `InputList` is the list of successfully matched and read items.

  `{error, What}` The read operation failed and the parameter `What` can be used as argument to `report_error/1` to produce an error message.

  `eof` End of file was encountered.

Examples:

```
> io:fread('enter>', "~f~f~f").
enter>1.9 35.5e3 15.0
{ok, [1.90000, 3.55000e+4, 15.0000]}
> io:fread('enter>', "~10f~d").
enter>     5.67899
{ok, [5.67800, 99]}
> io:fread('enter>', ":~10s:~10c:").
enter>:   alan  :   joe   :
{ok, ["alan", "   joe    "]}
```

`scan_erl_exprs(Prompt)`
`scan_erl_exprs([IoDevice,] Prompt, StartLine)`

Reads data from the standard input (`IoDevice`), prompting it with `Prompt`. Reading starts at line number `StartLine` (1). The data is tokenized as if it were a sequence of Erlang expressions until a final `'.'` is reached. This token is also returned. It returns:

{`ok, Tokens, EndLine`} The tokenization succeeded.

{`error, ErrorInfo, EndLine`} An error occurred.

{`eof, EndLine`} End of file was encountered.

Example:

```
> io:scan_erl_exprs('enter>').
enter>abc(), "hey".
{ok,[{atom, 1, abc},{'(', 1}, {')', 1}, {', ', 1},
    {string, 1, "hey"}, {dot, 1}], 2}
> io:scan_erl_exprs('enter>').
enter>1.0er.
{error, {1, erl_scan, float}, 2}
```

`scan_erl_form(Prompt)`
`scan_erl_form(IoDevice, Prompt[, StartLine])`

Reads data from the standard input (`IoDevice`), prompting it with `Prompt`. Starts reading at line number `StartLine` (1). The data is tokenized as if it were an Erlang form - one of the valid Erlang expressions in an Erlang source file - until a final `'.'` is reached. This last token is also returned. The return values are the same as for `scan_erl_exprs`.

`parse_erl_exprs(Prompt)`
`parse_erl_exprs(IoDevice, Prompt[, StartLine])`

Reads data from the standard input (`IoDevice`), prompting it with `Prompt`. Starts reading at line number `StartLine` (1). The data is tokenized and parsed as if it were a sequence of Erlang expressions until a final `'.'` is reached. It returns:

{`ok, ExpressionList, EndLine`} The parsing was successful.

{`error, ErrorInfo, EndLine`} An error occurred.

{`eof, EndLine`} End of file was encountered.

Example:

```
> io:parse_erl_exprs('enter>').
enter>abc(), "hey".
{ok, [{call, 1, [], abc, []}, {string, 1, "hey"}], 2}
> io:parse_erl_exprs ('enter>').
enter>abc("hey".
{error, {1, erl_parse, {before, {terminator,') '}, {dot, 1}}}, 2}
```

`parse_erl_form(Prompt)`
`parse_erl_form(IoDevice, Prompt[, StartLine])`

Reads data from the standard input (`IoDevice`), prompting it with `Prompt` Starts reading at line number `StartLine` (1). The data is tokenized and parsed as if it were an Erlang form - one of the valid Erlang expressions in an Erlang source file - until a final `'.'` is reached. It returns:

`{ok, Form, EndLine}` The parsing was successful.

`{error, ErrorInfo, EndLine}` An error occurred.

`{eof, EndLine}` End of file was encountered.

Standard Input/Output

All Erlang processes have a default standard IO device. This device is used when no `IoDevice` argument is specified in the IO calls. However, it is sometimes desirable to use an explicit `IoDevice` argument which refers to the default IO device. This is the case with functions that can access either a file or the default IO device. The atom `standard_io` has this special meaning. The following example illustrates this:

```
> io:read('enter>').
enter>foo.
{term, foo}
> io:read(standard_io, 'enter>').
enter>bar.
{term, bar}
```

There is always a process registered under the name of `user`. This can be used for sending output to the user.

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

io_lib (Module)

This module contains functions for converting to and from strings (lists of characters). They are used for implementing the functions in the `io` module. There is no guarantee that the character lists returned from some of the functions are flat, they can be deep lists. `lists:flatten/1` is used for generating flat lists.

Exports

`nl()`

Returns a character list which represents a new line character.

`write(Term)`
`write(Term, Depth)`

Returns a character list which represents `Term`. The `Depth` (-1) argument controls the depth of the structures written. When the specified depth is reached, everything below this level is replaced by "...". For example:

```
> lists:flatten(io_lib:write({1,[2],[3],[4,5],6,7,8,9})).
"{1,[2],[3],[4,5],6,7,8,9}"
> lists:flatten(io_lib:write({1,[2],[3],[4,5],6,7,8,9}, 5)).
"{1,[2],[3],[4|...],6|...}"
```

`print(Term)`
`print(Term, Column, LineLength, Depth)`

Also returns a list of characters which represents `Term`, but breaks representations which are longer than one line into many lines and indents each line sensibly. It also tries to detect and output lists of printable characters as strings. `Column` is the starting column (1), `LineLength` the maximum line length (80), and `Depth` the maximum print depth.

`fwrite(Format, Data)`
`format(Format, Data)`

Returns a character list which represents `Data` formatted in accordance with `Format`. Refer to io [page 112] for a detailed description of the available formatting options. A fault is generated if there is an error in the format string or argument list.

`fread(Format, String)`

Tries to read `String` in accordance with the control sequences in `Format`. Refer to io [page 112] for a detailed description of the available formatting options. It is assumed that `String` contains whole lines. It returns:

{`ok, InputList, LeftOverChars`} The string was read. `InputList` is the list of successfully matched and read items, and `LeftOverChars` are the input characters not used.

{`more, RestFormat, Nchars, InputStack`} The string was read, but more input is needed in order to complete the original format string. `RestFormat` is the remaining format string, `NChars` the number of characters scanned, and `InputStack` is the reversed list of inputs matched up to that point.

{`error,What`} An error occurred which can be formatted with the call `format_error/1`.

Example:

```
> io_lib:fread("~f~f~f", "15.6 17.3e-6 24.5").
{ok, [15.6000, 1.73000e-5, 24.5000], []}
```

`fread(Continuation, CharList, Format)`

This is the re-entrant formatted reader. It returns:

{`done, Result, LeftOverChars`} The input is complete. The result is one of the following:

{`ok, InputList`} The string was read. `InputList` is the list of successfully matched and read items, and `LeftOverChars` are the remaining characters.

`eof` End of file has been encountered. `LeftOverChars` are the input characters not used.

{`error,What`} An error occurred, which can be formatted with the call `format_error/1`.

{`more, Continuation`} More data is required to build a term. `Continuation` must be passed to <c>`fread/3`, when more data becomes available.

`write_atom(Atom)`

Returns the list of characters needed to print the atom `Atom`.

`write_string(String)`

Returns the list of characters needed to print `String` as a string.

`write_char(Integer)`

Returns the list of characters needed to print a character constant.

`indentation(String, StartIndent)`

Returns the indentation if `String` has been printed, starting at `Indentation`.

`char_list(CharList) -> bool()`

Returns `true` if CharList is a list of characters, otherwise it returns `false`.

`deep_char_list(CharList)`

Returns `true` if CharList is a deep list of characters, otherwise it returns `false`.

`printable_list(CharList)`

Returns `true` if CharList is a list of printable characters, otherwise it returns `false`.

Notes

The module `io_lib` also uses the extra modules `io_lib_format`, `io_lib_fread`, and `io_lib_pretty`. All external interfaces exist in `io_lib`.

Users are strongly advised not to access the other modules directly.

> **Note:**
> Any undocumented functions in `io_lib` should not be used.

The continuation of the first call to the re-entrant input functions must be `[]`. Refer to Armstrong, Virding, Williams, 'Concurrent Programming in Erlang', Chapter 13 for a complete description of how the re-entrant input scheme works

lib (Module)

The module `lib` provides the following useful library functions.

Exports

`flush_receive() -> void()`

Flushes the message buffer of the current process.

`error_message(Format, Args)`

Prints error message `Args` in accordance with `Format` in the normal way.

`progname() -> atom()`

Returns the name of the script that starts the current Erlang session.

`nonl(List1)`

Removes the last newline character, if any, in `List`.

`send(To, Msg)`

This function to makes it possible to send a message through `apply`.

`sendw(To, Msg)`

As `send/2`, but waits for an answer. It is implemented as follows:

```
sendw(To, Msg) ->
  To ! {self(),Msg},
  receive
    Reply -> Reply
  end.
```

The message returned is not necessarily a reply to the message sent.

Warning

This module is retained for compatibility. It may disappear without warning in a future release.

lists (Module)

This module contains functions for list processing. The functions are organized in two groups: those in the first group perform a particular operation on one ore several lists, whereas those in the second group perform use a user-defined function (given as the first argument) to perform an operation on one list.

Exports

`append(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns a list in which all the sub-lists of `ListOfLists` have been appended. For example:

```
> lists:append([[1, 2, 3], [a, b], [4, 5, 6]]).
[1, 2, 3, a, b, 4, 5, 6]
```

`append(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns a new list `List3` which is made from the elements of `List1` followed by the elements of `List2`. For example:

```
> lists:append("abc", "def").
"abcdef".
```

`lists:append(A,B)` is equivalent to `A ++ B`.

`concat(Things) -> string()`

Types:

- Things = [Thing]
- Thing = atom() | integer() | float() | string()

Concatenates the ASCII list representation of the elements of `Things`. The elements of `Things` can be atoms, integers, floats or strings.

```
> lists:concat([doc, '/', file, '.', 3]).
"doc/file.3"
```

`delete(Element, List1) -> List2`

Types:

- List1 = list2 = [Element]
- Element = term()

Returns a copy of `List1`, but the first occurrence of `Element`, if present, is deleted.

`duplicate(N, Element) -> List`

Types:

- N = int()
- List = [Element]
- Element = term()

Returns a list which contains N copies of the term `Element`.

> **Note:**
> `N` must be an integer >= 0. For example:

```
> lists:duplicate(5, xx).
[xx, xx, xx, xx, xx]
```

`flatlength(DeepList) -> int()`

Equivalent to `length(flatten(DeepList))`, but more efficient.

`flatten(DeepList) -> List`

Types:

- DeepList = [term() | DeepList]

Returns a flattened version of `DeepList`.

`flatten(DeepList, Tail) -> List`

Types:

- DeepList = [term() | DeepList]
- Tail = [term()]

Returns a flattened version of `DeepList` with the tail `Tail` appended.

`keydelete(Key, N, TupleList1) -> TupleList2`

Types:

- TupleList1 = TupleList2 = [tuple()]
- N = int()
- Key = term()

Returns a copy of `TupleList1` where the first occurrence of a tuple whose `N`th element is `Key` is deleted, if present.

`keymember(Key, N, TupleList) -> bool()`

Types:

- TupleList = [tuple()]
- N = int()
- Key = term()

Searches the list of tuples `TupleList` for a tuple whose `N`th element is `Key`.

`keymerge(N, List1, List2)`

Types:

- N = int()
- List1 = List2 = [tuple()]

Returns the sorted list formed by merging the `List1` and `List2`. The merge is performed on the `N`th element of each tuple. Both `List1` and `List2` must be key-sorted prior to evaluating this function; otherwise the order of the elements in the result will be undefined. When elements in the input lists compare equal, elements from `List1` are picked before elements from `List2`.

`keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2`

Types:

- Key = term()
- N = int()
- TupleList1 = TupleList2 = [tuple()]
- NewTuple = tuple()

Returns a list of tuples. In this list, a tuple is replaced by the tuple `NewTuple`. This tuple is the first tuple in the list where the element number `N` is equal to `Key`.

`keysearch(Key, N, TupleList) -> Result`

Types:

- TupleList = [tuple()]
- N = int()
- Key = term()
- Result = {value, tuple()} | false

Searches the list of the tuples `TupleList` for `Tuple` whose `N`th element is `Key`. Returns `{value, Tuple}` if such a tuple is found, or `false` if no such tuple is found.

`keysort(N, List1) -> List2`

Types:

- N = int()
- List1 = List2 = [tuple()]

Returns a list containing the sorted elements of `List1`. `TupleList1` must be a list of tuples, and the sort is performed on the `N`th element of the tuple. The sort is stable.

`last(List) -> Element`

Types:

- List = [Element]

- Element = term()

Returns the last element in `List`.

`max(List) -> Max`

Types:

- List = [Element]
- Element = Max = term()

Returns the maximum element of `List`.

`member(Element, List) -> bool()`

Types:

- List = [Element]
- Element = term()

Returns `true` if `Element` is contained in the list `List`, otherwise `false`.

`merge(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted prior to evaluating this function.

`merge(Fun, List1, List2) -> List`

Types:

- List = List1 = List2 = [Element]
- Fun = fun(Element, Element) -> bool()
- Element = term()

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted prior to evaluating this function, according to the ordering function `Fun`. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`min(List) -> Min`

Types:

- List = [Element]
- Element = Max = term()

Returns the minimum element of `List`.

`nth(N, List) -> Element`

Types:

- N = int()
- List = [Element]
- Element = term()

Returns the `N`th element of the `List`. For example:

```
> lists:nth(3, [a, b, c, d, e]).
c
```

`nthtail(N, List1) -> List2`

Types:

- N = int()
- List1 = List2 = [Alpha]

Returns the `N`th tail of `List`. For example:

```
> lists:nthtail(3, [a, b, c, d, e]).
[d, e]
```

`prefix(List1, List2) -> bool()`

Types:

- List1 = List2 = [term()]

Returns `true` if `List1` is a prefix of `List2`, otherwise `false`.

`reverse(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list with the top level elements in `List1` in reverse order.

`reverse(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns a list where `List1` has been reversed and appended to the beginning of `List2`. Equivalent to `reverse(List1) ++ List2`. For example:

```
> lists:reverse([1, 2, 3, 4], [a, b, c]).
[4, 3, 2, 1, a, b, c]
```

`seq(From, To) -> [int()]`
`seq(From, To, Incr) -> [int()]`

Types:

- From = To = Incr = int()

Returns a sequence of integers which starts with `From` and contains the successive results of adding `Incr` to the previous element, until `To` has been reached or passed (in the latter case, `To` is not an element of the sequence). If `To-From` has a different sign from `Incr`, or if `Incr` = 0 and `From` is different from `To`, an error is signalled (this implies that the result is never an empty list - the first element is always `From`).

`seq(From, To)` is equivalent to `seq(From, To, 1)`.

Examples:

```
> lists:seq(1, 10).
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

> lists:seq(1, 20, 3).
[1, 4, 7, 10, 13, 16, 19]

> lists:seq(1, 1, 0).
[1]
```

`sort(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list which contains the sorted elements of `List1`.

`sort(Fun, List1) -> List2`

Types:

- List1 = List2 = [Element]
- Fun = fun(Element, Element) -> bool()
- Element = term()

Returns a list which contains the sorted elements of `List1`, according to the ordering function `Fun`. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`sublist(List, N) -> List1`

Types:

- List1 = List2 = [term()]
- N = int()

Returns the first `N` elements of `List`. It is not an error for `N` to exceed the length of the list when `List` is a proper list - in that case the whole list is returned.

`sublist(List1, Start, Length) -> List2`

Types:

- List1 = List2 = [term()]
- Start = End = int()

Returns the sub-list of `List` starting at `Start` of length `Length`. Terminates with a runtime failure if `Start` is not in `List`, but a sub-list of a length less than `Length` is accepted. `Start` is considered to be in `List` if `Start` >= 1 and `Start` <= `length(List)+1`.

`subtract(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns a new list `List3` which is a copy of `List1`, subjected to the following procedure: for each element in `List2`, its first occurrence in `List1` is removed. For example:

```
> lists:subtract("123212", "212").
"312".
```

`lists:subtract(A,B)` is equivalent to `A -- B`.

`suffix(List1, List2) -> bool()`

Returns `true` if `List1` is a suffix of `List2`, otherwise `false`.

`sum(List) -> number()`

Types:

- List = [number()]

Returns the sum of the elements in `List`.

`all(Pred, List) -> bool()`

Types:

- Pred = fun(A) -> bool()
- List = [A]

Returns `true` if all elements `X` in `List` satisfy `Pred(X)`.

`any(Pred, List) -> bool()`

Types:

- Pred = fun(Element) -> bool()
- List = [Element]
- Element = term()

Returns `true` if any of the elements in `List` satisfies `Pred`.

`dropwhile(Pred, List1) -> List2`

Types:

- Pred = fun(A) -> bool()
- List1 = List2 = [A]

Drops elements `X` from `List1` while `Pred(X)` is `true` and returns the remaining list.

`filter(Pred, List1) -> List2`

Types:

- Pred = fun(A) -> bool()
- List1 = List2 = [A]

`List2` is a list of all elements `X` in `List1` for which `Pred(X)` is `true`.

`flatmap(Function, List1) -> Element`

Types:

- Function = fun(A) -> B
- List1 = [A]
- Element = [B]

`flatmap` behaves as if it had been defined as follows:

```
flatmap(Func, List) ->
    append(map(Func, List))
```

`foldl(Function, Acc0, List) -> Acc1`

Types:

- Function = fun(A, AccIn) -> AccOut
- List = [A]
- Acc0 = Acc1 = AccIn = AccOut = term()

`Acc0` is returned if the list is empty. For example:

```
> lists:foldl(fun(X, Sum) -> X + Sum end, 0, [1,2,3,4,5]).
15
> lists:foldl(fun(X, Prod) -> X * Prod end, 1, [1,2,3,4,5]).
120
```

`foldr(Function, Acc0, List) -> Acc1`

Types:

- Function = fun(A, AccIn) -> AccOut
- List = [A]
- Acc0 = Acc1 = AccIn = AccOut = term()

Calls `Function` on successive elements of `List` together with an extra argument `Acc` (short for accumulator). `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if the list is empty. `foldr` differs from `foldl` in that the list is traversed "bottom up" instead of "top down". `foldl` is tail recursive and would usually be preferred to `foldr`.

`foreach(Function, List) -> void()`

Types:

- Function = fun(A) -> void()
- List = [A]

Applies the function `Function` to each of the elements in `List`. This function is used for its side effects and the evaluation order is defined to be the same as the order of the elements in the list.

`map(Func, List1) -> List2`

Types:

- Func = fun(A) -> B
- List1 = [A]
- List2 = [B]

`map` takes a function from `As` to `Bs`, and a list of `As` and produces a list of `Bs` by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.

`mapfoldl(Function, Acc0, List1) -> {List2, Acc}`

Types:

- Function = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]

`mapfold` combines the operations of `map` and `foldl` into one pass. For example, we could sum the elements in a list and double them *at the same time*:

```
> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
                 0, [1,2,3,4,5]).
{[2,4,6,8,10],15}
```

`mapfoldr(Function, Acc0, List1) -> {List2, Acc}`

Types:

- Function = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]

`mapfold` combines the operations of `map` and `foldr` into one pass.

`splitwith(Pred, List) -> {List1, List2}`

Types:

- Pred = fun(A) -> bool()
- List = List1 = List2 = [A]

Partitions `Lists` into `List1` and `List2` according to `Pred`.

`splitwith` behaves as if it had been defined as follows:

```
splitwidth(Pred, List) ->
          {takewhile(Pred, List), dropwhile(Pred, List)}.
```

Note also that `List == List1 ++ List2`.

`takewhile(Pred, List1) -> List2`

Types:

- Pred = fun(A) -> bool()
- List1 = List2 = [A]

Returns the longest prefix of `List1` for which all elements `X` in `List1` satisfy `Pred(X)`.

Relics

Some of the exported functions in `lists.erl` are not documented. In particular, this applies to a number of `maps` and `folds` which have an extra argument for environment passing. These functions are no longer needed because Erlang 4.4 and later releases have *Funs*.

> **Note:**
> Any undocumented functions in lists should not be used.

log_mf_h (Module)

The `log_mf_h` is a `gen_event` handler module which can be installed in any `gen_event` process. It logs onto disk all events which are sent to an event manager. Each event is written as a binary which makes the logging very fast. However, a tool such as the `Report Browser` (`rb`) must be used in order to read the files. The events are written to multiple files. When all files have been used, the first one is re-used and overwritten. The directory location, the number of files, and the size of each file are configurable. The directory will include one file called `index`, and report files `1, 2, .....`

Exports

```
init(Dir, MaxBytes, MaxFiles)
init(Dir, MaxBytes, MaxFiles, Pred) -> Args
```

Types:

- Dir = string()
- MaxBytes = integer()
- MaxFiles = 0 < integer() < 256
- Pred = fun(Event) -> boolean()
- Event = term()
- Args = args()

Initiates the event handler. This function returns `Args`, which should be used in a call to `gen_event:add_handler(EventMgr, log_mf_h, Args)`.

`Dir` specifies which directory to use for the log files. `MaxBytes` specifies the size of each individual file. `MaxFiles` specifies how many files are used. `Pred` is a predicate function used to filter the events. If no predicate function is specified, all events are logged.

See Also

gen_event(3), rb(3)

math (Module)

This module provides an interface to a number of mathematical functions.

Exports

`pi() -> float()`

A useful number.

```
sin(X)
cos(X)
tan(X)
asin(X)
acos(X)
atan(X)
atan2(X, Y)
sinh(X)
cosh(X)
tanh(X)
asinh(X)
acosh(X)
atanh(X)
exp(X)
log(X)
log10(X)
pow(X, Y)
sqrt(X)
```

Types:

- X = Y = number()

A collection of math functions which return floats. Arguments are numbers.

`erf(X) -> float()`

Types:

- X = number()

Returns the error function of `X`, where

`erf(X) = 2/sqrt(pi)*integral from 0 to X of exp(-t*t) dt.`

`erfc(X) -> float()`

Types:

- X = number()

`erfc(X)` returns `1.0 - erf(X)`, computed by methods that avoid cancellation for large `X`.

Bugs

As these are the C library, the bugs are the same.

ordsets (Module)

Sets are collections of elements with no duplicate elements. An `ordset` is a representation of a set, where an ordered list is used to store the elements of the set. An ordered list is more efficient than an unordered list.

Exports

`new_set() -> OrdSet`

Types:

- OrdSet = ordset()

Returns a new empty ordered set.

`is_set(OrdSet) -> bool()`

Types:

- OrdSet = term()

Returns `true` if `OrdSet` is an ordered set of elements, otherwise `false`.

`set_to_list(OrdSet) -> List`

Types:

- OrdSet = ordset()
- List = [term()]

Returns the elements of `OrdSet` as a list.

`list_to_set(List) -> OrdSet`

Types:

- List = [term()]
- OrdSet = ordset()

Returns an ordered set of the elements in `List`.

`is_element(Element, OrdSet) -> bool()`

Types:

- Element = term()
- OrdSet = ordset()

Returns `true` if `Element` is an element of `OrdSet`, otherwise `false`.

`add_element(Element, OrdSet1) -> OrdSet2`

Types:

- Element = term()
- OrdSet1 = OrdSet2 = Ordset()

Returns a new ordered set formed from `OrdSet1` with `Element` inserted.

`del_element(Element, OrdSet1) -> OrdSet2`

Types:

- Element = term()
- OrdSet1 = OrdSet2 = ordset()

Returns `OrdSet1`, but with `Element` removed.

`union(OrdSet1, OrdSet2) -> OrdSet3`

Types:

- OrdSet1 = OrdSet2 = OrdSet3 = ordset()

Returns the merged (union) set of `OrdSet1` and `OrdSet2`.

`union(OrdSetList) -> OrdSet`

Types:

- OrdSetList = [ordset()]
- OrdSet = ordset()

Returns the merged (union) set of the list of sets.

`intersection(OrdSet1, OrdSet2) -> OrdSet3`

Types:

- OrdSet1 = OrdSet2 = OrdSet3 = ordset()

Returns the intersection of `OrdSet1` and `OrdSet2`.

`intersection(OrdSetList) -> OrdSet`

Types:

- OrdSetList = [ordset()]
- OrdSet = ordset()

Returns the intersection of the list of sets.

`subtract(OrdSet1, OrdSet2) -> OrdSet3`

Types:

- OrdSet1 = OrdSet2 = OrdSet3 = ordset()

Returns only the elements of `OrdSet1` which are not also elements of `OrdSet2`.

`subset(OrdSet1, OrdSet2) -> bool()`

Types:

- OrdSet1 = OrdSet2 = ordset()

Returns `true` when every element of `OrdSet1` is also a member of `OrdSet2`, otherwise `false`.

pg (Module)

This (experimental) module implements process groups. A process group is a group of processes that can be accessed by a common name. For example, a group named `foobar` can include a set of processes as members of this group and they can be located on different nodes.

When messages are sent to the named group, all members of the group receive the message. The messages are serialized. If the process `P1` sends the message `M1` to the group, and process `P2` simultaneously sends message `M2`, then all members of the group receive the two messages in the same order. If members of a group terminate, they are automatically removed from the group.

This module is not complete. The module is inspired by the ISIS system and the causal order protocol of the ISIS system should also be implemented. At the moment, all messages are serialized by sending them through a group master process.

Exports

`create(PgName)`

Creates an empty group named `PgName` on the current node.

`create(PgName, Node)`

Creates an epmty group on the node `Node`.

`join(PgName, Pid)`

Joins the Pid `Pid` to the process group `PgName`.

`send(Pgname, Message)`

Sends the tuple {`pg_message, From, PgName, Message`} to all members of the process group.

`esend(PgName, Mess)`

Sends the tuple {`pg_message, From, PgName, Message`} to all members of the process group, except the current node.

`members(PgName)`

Returns a list of the current members in the process group.

pool (Module)

`pool` can be used to run a set of Erlang nodes as a pool of computational processors. It is organized as a master and a set of slave nodes and includes the following features:

- The slave nodes send regular reports to the master about their current load.
- Queries can be sent to the master to determine which node will have the least load.

The BIF `statistics(run_queue)` is used for estimating future loads. It returns the length of the queue of ready to run processes in the Erlang system.

The slave nodes are started with the `slave` module. This effects, tty IO, file IO, and code loading.

If the master node fails, the entire pool will exit.

Exports

`start(Name)`

Starts a new pool. The file `.hosts.erlang` is read to find host names where the pool nodes can be started. The current working directory is searched first, then the home directory, and finally the root directory of the Erlang system. The start-up procedure fails if the file is not found.

`Name` is sent to all pool nodes. This is used as the first part of the node name in the `alive/3` statements for the nodes.

The function `net_adm:host_file()` reads the file `.hosts.erlang` for host names. The slave nodes are started with `slave:start`. See slave(3).

`start/1` is synchronous and all the nodes, as well as all the system servers, are running when it returns a value. Access rights must also be set so that all nodes in the pool have the authority to access each other.

`start(Name, Args)`

This function is the same as `start/1`, except that the environment `Args` is passed to the pool nodes. See slave(3).

`attach(Node)`

This function ensures that a pool master is running and includes `Node` in the pool master's pool of nodes.

`stop()`

Stops the pool and kills all the slave nodes.

`get_nodes()`

Returns a list of the current member nodes of the pool.

`pspawn(Mod, Fun, Args)`

Spawns a process on the pool node which is expected to have the lowest future load.

`pspawn_link(Mod, Fun, Args)`

Spawn links a process on the pool node which is expected to have the lowest future load.

`get_node()`

Returns the node ID of the node with the expected lowest future load.

`new_node(Host, Name)`

Starts a new node and attaches it to an already existing pool If there is no existing pool, it starts a pool with two nodes, the current node and `Node`. This function can also be used as a convenient way of starting new nodes, even if the load distribution facilities of `pool` are of no interest.

Files

$HOME/.hosts.erlang is used to pick hosts where nodes can be started.

$HOME/.erlang.slave.out.HOST is used for all additional IO that may come from the slave nodes on standard IO. If the start-up procedure does not work, this file may indicate the reason.

proc_lib (Module)

The `proc_lib` module is used to initialize some useful information when a process starts. The registered names, or the process identities, of the `parent` process, and the parent ancestors, are stored together with information about the function initially called in the process.

A crash report is generated if the process terminates with a reason other than `normal` or `shutdown`. `shutdown` is used to terminate an abnormal process in a controlled manner. A crash report contains the previously stored information such as ancestors and initial function, the termination reason, and information regarding other processes which terminate as a result of this process terminating.

The crash report is sent to the `error_logger`. An event handler has to be installed in the `error_logger` event manager in order to handle these reports. The crash report is tagged `crash_report` and the `format/1` function should be called in order to format the report.

Exports

```
spawn(Module,Func,Args) -> Pid
spawn(Node,Module,Func,Args) -> Pid
```

Types:

- Module = atom()
- Func = atom()
- Args = [Arg]
- Arg = term()
- Node = atom()
- Pid = pid()

Spawns a new process and initializes it as described above. The process is spawned using the `spawn` BIF. The process can be spawned on another `Node`.

```
spawn_link(Module,Func,Args) -> Pid
spawn_link(Node,Module,Func,Args) -> Pid
```

Types:

- Module = atom()
- Func = atom()
- Args = [Arg]
- Arg = term()
- Node = atom()

- Pid = pid()

Spawns a new process and initializes it as described above. The process is spawned using the `spawn_link` BIF. The process can be spawned on another `Node`.

```
start(Module,Func,Args) -> Ret
start(Module,Func,Args,Time) -> Ret
start_link(Module,Func,Args) -> Ret
start_link(Module,Func,Args,Time) -> Ret
```

Types:

- Module = atom()
- Func = atom()
- Args = [Arg]
- Arg = term()
- Time = integer >= 0 | infinity
- Ret = term() | {error, Reason}

Starts a new process synchronously. Spawns the process using `proc_lib:spawn/3` or `proc_lib:spawn_link/3`, and waits for the process to start. When the process has started, it *must* call `proc_lib:init_ack(Parent, Ret)` or `proc_lib:init_ack(Ret)`, where `Parent` is the process that evaluates `start`. At this time, `Ret` is returned from `start`.

If the `start_link` function is used and the process crashes before `proc_lib:init_ack` is called, `{error, Reason}` is returned if the calling process traps exits.

If `Time` is specified as an integer, this function waits for `Time` milliseconds for the process to start (`proc_lib:init_ack`). If it has not started within this time, `{error, timeout}` is returned, and the process is killed.

```
init_ack(Parent, Ret) -> void()
init_ack(Ret) -> void()
```

Types:

- Parent = pid()
- Ret = term()

This function is used by a process that has been started by a `proc_lib:start` function. It tells `Parent` that the process has initialized itself, has started, or has failed to initialize itself. The `init_ack/1` function uses the parent value previously stored by the `proc_lib:start` function. If the `init_ack` function is not called (e.g. if the init function crashes) and `proc_lib:start/3` is used, that function never returns and the parent hangs forever. This can be avoided by using a time out in the call to `start`, or by using `start_link`.

The following example illustrates how this function and `proc_lib:start_link` are used.

```
-module(my_proc).
-export([start_link/0]).
start_link() ->
    proc_lib:start_link(my_proc, init, [self()]).
init(Parent) ->
    case do_initialization() of
        ok ->
            proc_lib:init_ack(Parent, {ok, self()});
        {error, Reason} ->
            exit(Reason)
    end,
    loop().
loop() ->
    receive
        ....
```

`format(CrashReport) -> string()`

Types:

- CrashReport = void()

Formats a previously generated crash report. The formatted report is returned as a string.

`initial_call(PidOrPinfo) -> {Module,Function,Args} | false`

Types:

- PidOrPinfo = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = [void()]
- Module = atom()
- Function = atom()
- Args = [term()]

Extracts the initial call of a process that was spawned using the spawn functions described above. `PidOrPinfo` can either be a Pid, an integer tuple (from which a pid can be created), or the process information of a process (fetched through a `erlang:process_info/1` function call).

`translate_initial_call(PidOrPinfo) -> {Module,Function,Arity}`

Types:

- PidOrPinfo = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = [void()]
- Module = atom()
- Function = atom()
- Arity = int()

Extracts the initial call of a process which was spawned using the spawn functions described above. If the initial call is to one of the system defined behaviours such as `gen_server` or `gen_event`, it is translated to more useful information. If a `gen_server` is spawned, the returned `Module` is the name of the callback module and `Function` is `init` (the function that initiates the new server).

A `supervisor` and a `supervisor_bridge` are also `gen_server`processes. In order to return information that this process is a supervisor and the name of the call-back module, `Module` is `supervisor` and `Function` is the name of the supervisor callback module. `Arity` is `1` since the `init/1` function is called initially in the callback module.

By default, {`proc_lib,init_p,5`} is returned if no information about the initial call can be found. It is assumed that the caller knows that the process has been spawned with the `proc_lib` module.

`PidOrPinfo` can either be a Pid, an integer tuple (from which a pid can be created), or the process information of a process (fetched through a `erlang:process_info/1` function call).

This function is used by the c:I/0 and c:regs/0 functions in order to present process information.

See Also

error_logger(3)

queue (Module)

This module implements FIFO queues in an efficient manner.

Exports

`new() -> Queue`

Types:

- Queue = queue()

Returns an empty queue.

`in(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` into the queue `Q1`. Returns a new queue `Q2`.

`out(Q) -> Result`

Types:

- Result = {{value, Item}, Q1} | {empty, Q1}
- Q = Q1 = queue()

Removes the oldest element from the queue `Q`. Returns the tuple {{`value, Item`}, `Q1`}, where `Item` is the element removed and `Q1` is an identifier for the new queue. If `Q` is empty, the tuple {`empty, Q`} is returned.

`to_list(Q) -> list()`

Types:

- Q = queue()

Returns a list of the elements in the queue, with the oldest element first.

random (Module)

Random number generator. The method is attributed to B.A. Wichmann and I.D.Hill, in 'An efficient and portable pseudo-random number generator', Journal of Applied Statistics. AS183. 1982. Also Byte March 1987.

The current algorithm is a modification of the version attributed to Richard A O'Keefe in the standard Prolog library.

Exports

`seed() -> ran()`

Seeds random number generation with default (fixed) values.

`seed(A1, A2, A3) -> ran()`

Types:

- A1 = A2 = A3 = int()

Seeds random number generation with integer values.

`uniform() -> float()`

Returns a random float uniformly distributed between `0.0` and `1.0`.

`uniform(N) -> int()`

Types:

- N = int()

Given an integer `N >= 1`, `uniform(N)` returns a random integer uniformly distributed between `1` and `N`.

Note

Uses the process dictionary variable `random_seed` to remember the current seed.

Before a process calls `uniform/0` or `uniform/1` for the first time, it must call one of the seeding functions.

regexp (Module)

This module contains functions for regular expression matching and substitution.

Exports

`match(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()
- MatchRes = {match,Start,Length} | nomatch | {error,errordesc()}
- Start = Length = integer()

Finds the first, longest match of the regular expression `RegExp` in `String`. This function searches for the longest possible match and returns the first one found if there are several expressions of the same length. It returns as follows:

{`match,Start,Length`} if the match succeeded. `Start` is the starting position of the match, and `Length` is the length of the matching string.

`nomatch` if there were no matching characters.

{`error,Error`} if there was an error in `RegExp`.

`first_match(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()
- MatchRes = {match,Start,Length} | nomatch | {error,errordesc()}
- Start = Length = integer()

Finds the first match of the regular expression `RegExp` in `String`. This call is usually faster than `match` and it is also a useful way to ascertain that a match exists. It returns as follows:

{`match,Start,Length`} if the match succeeded. `Start` is the starting position of the match and `Length` is the length of the matching string.

`nomatch` if there were no matching characters.

{`error,Error`} if there was an error in `RegExp`.

`matches(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()
- MatchRes = {match, Matches} | {error, errordesc()}
- Matches = list()

Finds all non-overlapping matches of the expression `RegExp` in `String`. It returns as follows:

`{match, Matches}` if the regular expression was correct. The list will be empty if there was no match. Each element in the list looks like `{Start, Length}`, where `Start` is the starting position of the match, and `Length` is the length of the matching string.

`{error,Error}` if there was an error in `RegExp`.

`sub(String, RegExp, New) -> SubRes`

Types:

- String = RegExp = New = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

Substitutes the first occurrence of a substring matching `RegExp` in `String` with the string `New`. A `&` in the string `New` is replaced by the matched substring of `String`. `\&` puts a literal `&` into the replacement string. It returns as follows:

`{ok,NewString,RepCount}` if `RegExp` is correct. `RepCount` is the number of replacements which have been made (this will be either 0 or 1).

`{error, Error}` if there is an error in `RegExp`.

`gsub(String, RegExp, New) -> SubRes`

Types:

- String = RegExp = New = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

The same as `sub`, except that all non-overlapping occurrences of a substring matching `RegExp` in `String` are replaced by the string `New`. It returns:

`{ok,NewString,RepCount}` if `RegExp` is correct. `RepCount` is the number of replacements which have been made.

`{error, Error}` if there is an error in `RegExp`.

`split(String, RegExp) -> SplitRes`

Types:

- String = RegExp = string()
- SubRes = {ok,FieldList} | {error,errordesc()}
- Fieldlist = [string()]

`String` is split into fields (sub-strings) by the regular expression `RegExp`.

If the separator expression is `" "` (a single space), then the fields are separated by blanks and/or tabs and leading and trailing blanks and tabs are discarded. For all other values of the separator, leading and trailing blanks and tabs are not discarded. It returns:

`{ok, FieldList}` to indicate that the string has been split up into the fields of `FieldList`.

`{error, Error}` if there is an error in `RegExp`.

`sh_to_awk(ShRegExp) -> AwkRegExp`

Types:

- ShRegExp AwkRegExp = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

Converts the `sh` type regular expression `ShRegExp` into a full `AWK` regular expression. Returns the converted regular expression string. `sh` expressions are used in the shell for matching file names and have the following special characters:

`*` matches any string including the null string.

`?` matches any single character.

`[...]` matches any of the enclosed characters. Character ranges are specified by a pair of characters separated by a `-`. If the first character after `[` is a `!`, then any character not enclosed is matched.

It may sometimes be more practical to use `sh` type expansions as they are simpler and easier to use, even though they are not as powerful.

`parse(RegExp) -> ParseRes`

Types:

- RegExp = string()
- ParseRes = {ok,RE} | {error,errordesc()}

Parses the regular expression `RegExp` and builds the internal representation used in the other regular expression functions. Such representations can be used in all of the other functions instead of a regular expression string. This is more efficient when the same regular expression is used in many strings. It returns:

`{ok, RE}` if `RegExp` is correct and `RE` is the internal representation.

`{error, Error}` if there is an error in `RegExpString`.

`format_error(ErrorDescriptor) -> string()`

Types:

- ErrorDescriptor = errordesc()

Returns a string which describes the error `ErrorDescriptor` returned when there is an error in a regular expression.

Regular Expressions

The regular expressions allowed here is a subset of the set found in `egrep` and in the `AWK` programming language, as defined in the book, `The AWK Programming Language, by A. V. Aho, B. W. Kernighan, P. J. Weinberger`. They are composed of the following characters:

`c` matches the non-metacharacter `c`.

`\c` matches the escape sequence or literal character `c`.

`.` matches any character.

`^` matches the beginning of a string.

`$` matches the end of a string.

`[abc...]` character class, which matches any of the characters `abc...` Character ranges are specified by a pair of characters separated by a `-`.

`[^abc...]` negated character class, which matches any character except `abc...`.

`r1 | r2` alternation. It matches either `r1` or `r2`.

`r1r2` concatenation. It matches `r1` and then `r2`.

`r+` matches one or more `r`s.

`r*` matches zero or more `r`s.

`r?` matches zero or one `r`s.

`(r)` grouping. It matches `r`.

The escape sequences allowed are the same as for Erlang strings:

`\b` backspace

`\f` form feed

`\n` newline (line feed)

`\r` carriage return

`\t` tab

`\e` escape

`\v` vertical tab

`\s` space

`\d` delete

`\ddd` the octal value ddd

`\c` any other character literally, for example `\\` for backslash, `\"` for “)

To make these functions easier to use, in combination with the function `io:get_line` which terminates the input line with a new line, the `$` characters also matches a string ending with `"...\n"`. The following examples define Erlang data types:

```
Atoms      [a-z][0-9a-zA-Z_]*

Variables  [A-Z_][0-9a-zA-Z_]*

Floats     (\+|-)?[0-9]+\.[0-9]+((E|e)(\+|-)?[0-9]+)?
```

Regular expressions are written as Erlang strings when used with the functions in this module. This means that any \ or " characters in a regular expression string must be written with \ as they are also escape characters for the string. For example, the regular expression string for Erlang floats is:
`"(\\+|-)?[0-9]+\\.[0-9]+((E|e)(\\+|-)?[0-9]+)?".`

It is not really necessary to have the escape sequences as part of the regular expression syntax as they can always be generated directly in the string. They are included for completeness and can they can also be useful when generating regular expressions, or when they are entered other than with Erlang strings.

shell (Module)

The module `shell` implements an Erlang shell.

The shell is a user interface program for entering expression sequences. The expressions are evaluated and a value is returned. A history mechanism saves previous commands and their values, which can then be incorporated in later commands.

Variable bindings, and local process dictionary changes which are generated in user expressions, are preserved and the variables can be used in later commands to access their values. The bindings can also be forgotten so the variables can be re-used.

The special shell commands all have the syntax of (local) function calls. They are evaluated as normal function calls and many commands can be used in one expression sequence.

If a command (local function call) is not recognized by the shell, an attempt is first made to find the function in the module `user_default`, where customized local commands can be placed. If found, then the function is evaluated. Otherwise, an attempt is made to evaluate the function in the module `shell_default`. The module `user_default` must be explicitly loaded.

The shell also permits the user to start multiple concurrent jobs. A job can be regarded as a set of processes which can communicate with the shell.

The shell runs in two modes:

- `Normal` mode, in which commands can be edited and expressions evaluated
- Job Control Mode `JCL`, in which jobs can be started, killed, detached and connected.

Only the currently connected job can 'talk' to the shell.

Shell Commands

`b()` Prints the current variable bindings.

`f()` Removes all variable bindings.

`f(X)` Removes the binding of variable `X`.

`h()` Prints the history list.

`e(N)` Repeats the command `N`, if `N` is positive. If it is negative, the `N`th previous command is repeated (i.e. `e(-1)` repeats the previous command).

`v(N)` Uses the return value of the command `N` in the current command.

`help()` Evaluates `shell_default:help()`.

`c(File)` Evaluates `shell_default:c(File)`. This compiles and loads code in `File` and purges old versions of code, if necessary. Assumes that the file and module names are the same.

Example

The following example is a long dialogue with the shell. Commands starting with > are inputs to the shell. All other lines are output from the shell. All commands in this example are explained at the end of the dialogue. .

```
strider 1> erl
Erlang (BEAM) emulator version 4.9

Eshell V4.9  (abort with ^G)
1> Str = "abcd".
"abcd"
2> L = length(Str).
4
3> Descriptor = {L, list_to_atom(Str)}.
{4,abcd}
4> L.
4
5> b().
Descriptor = {4,abcd}
L = 4
Str = "abcd"
ok
6> f(L).
ok
7> b().
Descriptor = {4,abcd}
Str = "abcd"
ok
8> f(L).
ok
9> {L, _} = Descriptor.
{4,abcd}
10> L.
4
11> {P, Q, R} = Descriptor.
** exited: {{badmatch,{4,abcd}},{erl_eval,expr,3}} **
12> P.
** exited: {{unbound,'P'},{erl_eval,expr,3}} **
13> Descriptor.
{4,abcd}
14> {P, Q} = Descriptor.
{4,abcd}
15> P.
4
16> f().
ok
17> put(aa, hello).
undefined
18> get(aa).
hello
19> Y = test1:demo(1).
```

```
11
20> get().
[{aa,worked}]
21> put(aa, hello).
worked
22> Z = test1:demo(2).
** exited: {{badmatch,1},{test1,demo,[2]}} **

=ERROR REPORT==== 24-Jan-1997::07:48:46 ===
!!! Error in process <0.22.0> with exit value: {{badmatch,1}
,{test1,demo,[2]}}
23> Z.
** exited: {{unbound,'Z'},{erl_eval,expr,3}} **
24> get(aa).
hello
25> erase(), put(aa, hello).
undefined
26> spawn(test1, demo, [1]).
<0.25.0>
27> get(aa).
hello
28> io:format("hello hello\n").
hello hello
ok
29> e(28).
hello hello
ok
30> v(28).
ok
31> test1:loop(0).
Hello Number: 0
Hello Number: 1
Hello Number: 2
Hello Number: 3

User switch command
 --> i
 --> c
.
.
.
Hello Number: 3374
Hello Number: 3375
Hello Number: 3376
Hello Number: 3377
Hello Number: 3378
** exited: killed **
32> halt().
strider 2>
```

Comments

Command 1 sets the variable `Str` to the string `"abcd"`.

Command 2 sets `L` to the length of the string evaluating the BIF `atom_to_list`.

Command 3 builds the tuple `Descriptor`.

Command 4 prints the value of the variable `L`.

Command 5 evaluates the internal shell command `b()`, which is an abbreviation of "bindings". This prints the current shell variables and their bindings. The `ok` at the end is the return value of the `b()` function.

Command 6 `f(L)` evaluates the internal shell command `f(L)` (abbreviation of "forget"). The value of the variable `L` is removed.

Command 7 prints the new bindings.

Command 8 shows that the value of `L` has disappeared from the bindings.

Command 9 performs a pattern matching operation on `Descriptor`, binding a new value to `L`.

Command 10 prints the current value of `L`.

Command 11 tries to match {`P, Q, R`} against `Descriptor` which is {`4, abc`}. The match fails and none of the new variables become bound. The printout starting with "`** exited:`" is not the value of the expression (the expression had no value because its evaluation failed), but rather a warning printed by the system to inform the user that an error has occurred. The values of the other variables (`L`, `Str`, etc.) are unchanged.

Commands 12 and 13 show that `P` is unbound because the previous command failed, and that `Descriptor` has not changed.

Commands 14 and 15 show a correct match where `P` and `Q` are bound.

Command 16 clears all bindings.

The next few commands assume that `test1:demo(X)` is defined in the following way:

```
demo(X) ->
    put(aa, worked),
    X = 1,
    X + 10.
```

Commands 17 and 18 set and inspect the value of the item `aa` in the process dictionary.

Command 19 evaluates `test1:demo(1)`. The evaluation succeeds and the changes made in the process dictionary become visible to the shell. The new value of the dictionary item `aa` can be seen in command 20.

Commands 21 and 22 change the value of the dictionary item `aa` to `hello` and call `test1:demo(2)`. Evaluation fails and the changes made to the dictionary in `test1:demo(2)`, before the error occurred, are discarded.

Commands 23 and 24 show that `Z` was not bound and that the dictionary item `aa` has retained its original value.

Commands 25, 26 and 27 show the effect of evaluating `test1:demo(1)` in the background. In this case, the expression is evaluated in a newly spawned process. Any changes made in the process dictionary are local to the newly spawned process and therefore not visible to the shell.

Commands 28, 29 and 30 use the history facilities of the shell.

Command 29 is `e(28)`. This re-evaluates command 28. Command 30 is `v(28)`. This uses the value (result) of command 28. In the cases of a pure function (a function with no side effects), the result is the same. For a function with side effects, the result can be different.

For the next command, it is assumed that `test1:loop(N)` is defined in the following way:

```
loop(N) ->
    io:format("Hello Number: ~w~n", [N]),
    loop(N+1).
```

Command 31 evaluates `test1:loop(0)`, which puts the system into an infinite loop. At this point the user types `Control G`, which suspends output from the current process, which is stuck in a loop, and activates `JCL` mode. In `JCL` mode the user can start and stop jobs.

In this particular case, the `i` command ("interrupt") is used to terminate the looping program, and the `c` command is used to connect to the shell again. Since the process was running in the background before we killed it, there will be more printouts before the "`** exited: killed **`" message is shown.

The `halt()` command exits the Erlang system.

JCL Mode

When the shell starts, it starts a single evaluator process. This process, together with any local processes which it spawns, is referred to as a `job`. Only the current job, which is said to be `connected`, can perform operations with standard IO. All other jobs, which are said to be `detached`, are `blocked` if they attempt to use standard IO.

All jobs which do not use standard IO run in the normal way.

`^G` (Control G) detaches the current job and JCL mode is activated. The `JCL` mode prompt is `"-->"`. If `"?"` is entered at the prompt, the following help message is displayed:

```
--> ?
c [nn]   - connect to job
i [nn]   - interrupt job
k [nn]   - kill job
j        - list all jobs
s        - start local shell
r [node] - start remote shell
q        - quit Erlang
?  | h   - this message
```

The `JCL` commands have the following meaning:

`c [nn]` Connects to job number <`nn`> or the current job. The standard shell is resumed. Operations which use standard IO by the current job will be interleaved with user inputs to the shell.

`i [nn]` Stops the current evaluator process for job number `nn` or the current job, but does not kill the shell process. Accordingly, any variable bindings and the process dictionary will be preserved and the job can be connected again. This command can be used to interrupt an endless loop.

`k [nn]` Kills job number `nn` or the current job. All spawned processes in the job are killed, provided they have not evaluated the `group_leader/1` BIF and are located on the local machine. Processes spawned on remote nodes will not be killed.

`j` Lists all jobs. A list of all known jobs is printed. The current job name is prefixed with '*'.

`s` Starts a new job. This will be assigned the new index `[nn]` which can be used in references.

`r [node]` Starts a remote job on `node`. This is used in distributed Erlang to allow a shell running on one node to control a number of applications running on a network of nodes.

`q` Quits Erlang.

`?` Displays this message.

Bugs

There is no way of changing the length of the history list or saving it between sessions.

shell_default (Module)

The functions in `shell_default` are called when no module name is given in a shell command.

Consider the following shell dialogue:

```
1 > lists:reverse("abc").
"cab"
2 > c(foo).
{ok, foo}
```

In command one, the module `lists` is called. In command two, no module name is specified. The shell searches the modules `user_default` followed by `shell_default` for the function `foo/1`.

`shell_default` is intended for "system wide" customizations to the shell. `user-default` is intended for "local" or individual user customizations.

Hint

To add your own commands to the shell, create a module called `user_default` and add the commands you want. Then add the following line as the *first* line in your `.erlang` file in your home directory.

```
code:load_abs("$PATH/user_default").
```

`$PATH` is the directory where your `user_default` module can be found.

slave (Module)

This module provides functions for starting Erlang slave nodes. All slave nodes which are started by a master will terminate automatically when the master terminates. All TTY output produced at the slave will be sent back to the master node. File I/O is done via the master.

Slave nodes on other hosts than the current one are started with the program `rsh`. The user must be allowed to `rsh` to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the `rsh` documentation for details). A slave node started on the same host as the master inherits certain environment values from the master, such as the current directory and the environment variables. For what can be assumed about the environment when a slave is started on another host, read the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to `erl` as follows: `-rsh Program`.

The slave node should use the same file system at the master. At least, the Erlang system should be installed in the same place on both computers and the same version of Erlang should be used.

Currently, a node running on Windows NT can only start slave nodes on the host on which it is running.

The master node must be alive.

Exports

`start(Host)`

Starts a slave node on the host `Host`. Host names need not necessarily be specified as fully qualified names; short names can also be used. This is the same condition that applies to names of distributed Erlang nodes. The name of the started node will be the same as the node which executes the call, with the exception of the host name part of the node name.

Return value: see `start/3`.

`start_link(Host)`

Starts a slave node on the host `Host` in the same way as the `start/1`, except that the slave node is linked to the currently executing process. If the process terminates, the slave node also terminates.

Return value: see `start/3`.

`start(Host, Name)`

Starts a slave node on the host `Host` with the name `Name@Host`.

Return value: see `start/3`.

`start_link(Host, Name)`

Starts a slave node on the host `Host` in the same way as `start/2`, except that the slave node is linked to the currently executing process. If that process terminates, the slave node also terminates.

Return value: see `start/3`.

`start(Host, Name, Args) -> {ok, Node} | {error, ErrorInfo}`

Starts a slave node with the name `Name@Host` on `Host` and passes the argument string `Args` to the new node.

The slave node resets its `user` process so that all terminal I/O which is produced at the slave is automatically relayed to the master. Also, the file process will be relayed to the master.

The `Args` argument can be used for a variety of purposes. See erl(1). For example, the following command line arguments can be passed to the slave:

- to set some environment variable on the slave
- to run some specific program on the slave
- to set some specific code path on the slave node.

As an example, suppose that we want to start a slave node at host `H` with the node name `Name@H`, and we also want the slave node to have the following properties:

- directory `Dir` should be added to the code path;
- the Mnesia directory should be set to `M`;
- the unix DISPLAY environment variable should be set to the display of the master node.

The following code is executed to achieve this:

```
E = " -env DISPLAY " ++ net_adm:localhost() ++ ":0 ",
Arg = "-mnesia_dir " ++ M ++ " -pa " ++ Dir ++ E,
slave:start(H, Name, Arg).
```

The `start/3` call returns `{ok, Name@Host}` if successful, otherwise `{error, Reason}`. `Reason` can be one of:

`timeout` The master node failed to get in contact with the slave node. This can happen in a number of circumstances:

- there is no Erlang system installed on the remote host
- the file system on the other host has a different structure to the the master
- the Erlang systems have different cookies.

`no_rsh` There is no `rsh` program on the computer.

`{already_running, Name@Host}` A node with the name `Name@Host` already exists.

`start_link(Host, Name, Args)`

Starts a slave node on the host `Host` in the same way as the `start/3`, except that the slave node is linked to the currently executing process. If that process terminates, the slave node also terminates.

Return value: see `start/3`.

`stop(Node)`

Stops (kills) a node.

`pseudo([Master | ServerList])`

Calls `pseudo(Master, ServerList)`. If we want to start a node from the command line and set up a number of pseudo servers, an Erlang system can be started as follows:

```
% erl -name abc -s slave pseudo klacke@super x --
```

`pseudo(Master, ServerList)`

Starts a number of pseudo servers. A pseudo server is a server with a registered name which does absolutely nothing but pass on all message to the real server which executes at a master node. A pseudo server is an intermediary which only has the same registered name as the real server.

For example, if we have started a slave node `N` and want to execute `pxw` graphics code on this node, we can start the server `pxw_server` as a psudo server at the slave node. The following code illustrates:

```
rpc:call(N, slave, pseudo, [node(), [pxw_server]]).
```

`relay(Pid)`

Runs a pseudo server. This function never returns any value and the process which executes the function will receive messages. All messages received will simply be passed on to `Pid`.

string (Module)

This module contains functions for string processing.

Exports

`len(String) -> Length`

Types:

- String = string()
- Length = integer()

Returns the number of characters in the string.

`equal(String1, String2) -> bool()`

Types:

- String1 = String2 = string()

Tests whether two strings are equal. Returns `true` if they are, otherwise `false`.

`concat(String1, String2) -> String3`

Types:

- String1 = String2 = String3 = string()

Concatenates two strings to form a new string. Returns the new string.

`chr(String, Character) -> Index`
`rchr(String, Character) -> Index`

Types:

- String = string()
- Character = char()
- Index = integer()

Returns the index of the first/last occurrence of `Character` in `String`. `0` is returned if `Character` does not occur.

`str(String, SubString) -> Index`
`rstr(String, SubString) -> Index`

Types:

- String = SubString = string()

- Index = integer()

Returns the position where the first/last occurence of `SubString` begins in `String`. `0` is returned if `SubString` does not exist in `String`. For example:

```
> string:str(" Hello Hello World World ", "Hello World").
8
```

`span(String, Chars) -> Length`
`cspan(String, Chars) -> Length`

Types:

- String = Chars = string()
- Length = integer()

Returns the length of the maximum initial segment of String, which consists entirely of characters from (not from) Chars.

For example:

```
> string:span("\t    abcdef", " \t").
5
> string:cspan("\t    abcdef", " \t").
0
```

`substr(String, Start) -> SubString`
`substr(String, Start, Length) -> Substring`

Types:

- String = SubString = string()
- Start = Length = integer()

Returns a substring of `String`, starting at the position `Start`, and ending at the end of the string or at length `Length`.

For example:

```
> substr("Hello World", 4, 5).
"lo Wo"
```

`tokens(String, SeperatorList) -> Tokens`

Types:

- String = SeperatorList = string()
- Tokens = [string()]

Returns a list of tokens in `String`, separated by the characters in `SeperatorList`.

For example:

```
> tokens("abc defxxghix jkl", "x ").
["abc", "def", "ghi", "jkl"]
```

`chars(Character, Number) -> String`
`chars(Character, Number, Tail) -> String`

Types:

- Character = char()

- Number = integer()
- String = string()

Returns a string consisting of `Number` of characters `Character`. Optionally, the string can end with the string `Tail`.

`copies(String, Number) -> Copies`

Types:

- String = Copies = string()
- Number = integer()

Returns a string containing `String` repeated `Number` times.

`words(String) -> Count`
`words(String, Character) -> Count`

Types:

- String = string()
- Character = char()
- Count = integer()

Returns the number of words in `String`, separated by blanks or `Character`.

For example:

```
> words(" Hello old boy!", $o).
4
```

`sub_word(String, Number) -> Word`
`sub_word(String, Number, Character) -> Word`

Types:

- String = Word = string()
- Character = char()
- Number = integer()

Returns the word in position `Number` of `String`. Words are separated by blanks or `Characters`.

For example:

```
> string:sub_word(" Hello old boy !",3,$o).
"ld b"
```

`strip(String) -> Stripped`
`strip(String, Direction) -> Stripped`
`strip(String, Direction, Character) -> Stripped`

Types:

- String = Stripped = string()
- Direction = left | right | both
- Character = char()

Returns a string, where leading and/or trailing blanks or a number of `Character` have been removed. `Direction` can be `left`, `right`, or `both` and indicates from which direction blanks are to be removed. The function `strip/1` is equivalent to `strip(String, both)`.

For example:

```
> string:strip("...Hello.....", both, $.).
"Hello"
```

`left(String, Number) -> Left`
`left(String, Number, Character) -> Left`

Types:

- String = Left = string()
- Character = char
- Number = integer()

Returns the `String` with the length adjusted in accordance with `Number`. The left margin is fixed. If the `length(String)` < `Number`, `String` is padded with blanks or `Characters`.

For example:

```
> string:left("Hello",10,$.).
"Hello....."
```

`right(String, Number) -> Right`
`right(String, Number, Character) -> Right`

Types:

- String = Right = string()
- Character = char
- Number = integer()

Returns the `String` with the length adjusted in accordance with `Number`. The right margin is fixed. If the length of `(String)` < `Number`, `String` is padded with blanks or `Characters`.

For example:

```
> string:right("Hello", 10, $.).
".....Hello"
```

`centre(String, Number) -> Centered`
`centre(String, Number, Character) -> Centered`

Types:

- String = Centered = string()
- Character = char
- Number = integer()

Returns a string, where `String` is centred in the string and surrounded by blanks or characters. The resulting string will have the length `Number`.

`sub_string(String, Start) -> SubString`

`sub_string(String, Start, Stop) -> SubString`

Types:

- String = SubString = string()
- Start = Stop = integer()

Returns a substring of `String`, starting at the position `Start` to the end of the string, or to and including the `Stop` position.

For example:

```
sub_string("Hello World", 4, 8).
"lo Wo"
```

Notes

Some of the general string functions may seem to overlap each other. The reason for this is that this string package is the combination of two earlier packages and all the functions of both packages have been retained.

The regular expression functions have been moved to their own module `regexp` (see regexp [page 148]). The old entry points still exist for backwards compatibility, but will be removed in a future release so that users are encouraged to use the module `regexp`.

> **Note:**
> Any undocumented functions in `string` should not be used.

supervisor (Module)

A *supervisor* is a process that supervises *child* processes. A child can be another supervisor or a *worker* process. A supervisor is always linked to its children. This structure is used to build a *supervision tree*, which is a nice way to structure an application for fault tolerance.

The basic idea of a supervisor is that it keeps its children alive. If a child terminates abnormally, it is restarted. There are three basic types of restart strategies for supervisors, *one-for-one*, *one-for-all*, and *rest-for-one*

- If a child in a one-for-one supervisor dies abnormally, it is restarted.
- If a child in a one-for-all supervisor dies, the supervisor shuts down all of the other children and then restarts all children. This strategy can be used when there are dependencies among the children.
- If a child in a rest-for-one supervisor dies, all children started *after* the faulty child are shut down, then restarted. The children started before the faulty child are not affected.

There is yet another restart strategy which is a variant of the ordinary one-for-one. It is called *simple-one-for-one*. It should be used for dynamic processes of the same type, for example processes which represent a call. Compared to one-for-one, this type has reduced overheads in starting dynamic children .

Each child can be one of three types: *permanent*, *transient*, or *temporary*. A permanent child is always restarted when it dies. A transient child is restarted if it dies abnormally, and a temporary child is never restarted.

The supervisors have a built-in mechanism to prevent situations where a child dies, is restarted by the supervisor, only to die again for the same reason, is restarted again, and so on. It limits the number of restarts which can occur in a given time interval. This is determined by the values of two parameters, `MaxR` and `MaxT`. If more than `MaxR` restarts are performed in the last `MaxT` seconds, then the supervisor shuts down all the children which it supervises and then dies.

An instance of the supervisor behaviour can be debugged using the module `sys`.

Exports

```
start_link(Module,StartArgs) -> SupRet
start_link(SupName,Module,StartArgs) -> SupRet
```

Types:

- SupName = {local, atom()} | {global, atom()}
- Module = atom()

- StartArgs = term()
- SupRet = {ok, Pid} | ignore | {error, Reason}
- Pid = pid()
- Reason = {already_started, Pid} | term()

Starts a new instance of the supervisor behaviour. The function `Module:init(StartArgs)` is called in order to create a start specification (see below).

If the supervisor is started without `SupName`, it can only be called using the returned `Pid` identifier. If it is started with `SupName`, the name is registered locally or globally.

`start_child(Supervisor,ChildSpec | ExtraStartArgs) -> {ok, Child} | {ok, Child, Info} | {error, Reason}`

Types:

- Supervisor = pid() | SupName | {global, SupName}
- ChildSpec = child_spec()
- ExtraStartArgs = [term()]
- child_spec() = {Name, Start, Restart, Shutdown, Type, Modules}
- SupName = atom()
- Name = term()
- Start = {M, F, A}
- Restart = permanent | transient | temporary
- Shutdown = int() >= 0 | brutal_kill | infinity
- Type = worker | supervisor
- Modules = [atom()] | dynamic
- Child = pid() | undefined
- Info = term()

Use this function to dynamically add a child to a supervisor. The start function `Start` is supposed to return `{ok, Pid} | {ok, Pid, Info} | ignore | {error, Reason}`. If `ignore` is returned, the supervisor ignores the child and returns `{ok, undefined}`. The start function is executed by the supervisor process. It must return a Pid that is linked to the caller (i.e. the supervisor). The supervisor uses this link to monitor and control the child. If `{ok, Pid, Info}` is returned from the start function, the same is returned from this function. The `Info` is not interpreted in any way by the supervisor.

`Name` is an internal name, which is used by the supervisor to identify its children.

`Modules` is used for the code change procedure. It should be `dynamic` if the modules that the child uses can change dynamically at runtime, for example a `gen_event` process. (Note that this refers to the names of the modules rather than the implementation of the module.) Otherwise, it should be a list of the module with which the child is implemented, This information is used by the release handler to find all processes which execute a module. For example, if the child is a `gen_server`, `Modules` is a list with the name of the callback module as its only element.

The `Shutdown` value `infinity` must be used with care. The supervisor tries to shut down the child by calling `exit(Child, shutdown)` and waits for the child to terminate. If the child does not terminate, the supervisor will hang forever. `infinity` should be used for children which themselves are supervisors, but it is not allowed for workers. This is to make sure that the system can be shut down without hanging forever.

If the supervisor is a `simple_one_for_one` supervisor, this function should be called as `start_child(Supervisor, ExtraStartArgs)`. It starts a new child of the same type

and calls the child's start function as `apply(M, F, A ++ ExtraStartArgs)`. `M`, `F`, and `A` are returned from the supervisor's `init` function. The new child does not get a unique name by which is identified in the supervisor. Therefore, the functions `terminate_child/2`, `delete_child/2` and `restart_child/2` cannot be used for a `simple_one_for_one` supervisor. When a `temporary` child dies for any reason or a `transient` child dies normally, the child is removed from the supervisor. Compare this with a ordinary supervisor, where the child specification remains until `delete_child/2` is called. No progress report is generated when the child is started. This is to reduce overheads.

`terminate_child(Supervisor, Name) -> ok | {error, not_found}`

Types:

- Supervisor = pid() | SupName | {global, SupName}
- SupName = atom()
- Name = term()

Terminates a child. The child is not removed from the supervisor's set of children. This means that it can be restarted explicitly by calling `restart_child/2`, or started implicitly if the supervisor has to restart all children.

`delete_child(Supervisor,Name) -> ok | {error, running | not_found}`

Types:

- Supervisor = pid() | SupName | {global, SupName}
- SupName = atom()
- Name = term()

Deletes a child from the supervisor. The child must be terminated.

`restart_child(Supervisor,Name) -> {ok, Pid} | {ok, Pid, Info} | {error, running | not_found | Reason}`

Types:

- Supervisor = pid() | SupName | {global, SupName}
- SupName = atom()
- Name = term()
- Info = term()

Starts a child which has been terminated and not restarted according to the restart specification. This can include a temporary child which terminates, or a child that was terminated explicitly by calling the function `terminate_child/2`.

`which_children(Supervisor) -> [{Name, Pid, Type, Modules}]`

Types:

- Supervisor = pid() | SupName | {global, SupName}
- SupName = atom()
- Name = term()
- Pid = pid() | undefined
- Type = worker | supervisor
- Modules = [atom()] | dynamic

Returns a list of the supervisor's children. `Name`, `Type` and `Modules` are as defined in the child specification.

`check_childspecs([ChildSpec]) -> ok | {error, Reason}`

Types:

- ChildSpec = child_spec()

Checks if a list of child specifications are syntactically correct.

Callback Functions

The following functions should be exported from a `supervisor` callback module.

Exports

`Module:init(StartArgs) -> {ok, {SupFlags, [ChildSpec]}} | ignore | {error, Reason}`

Types:

- SupFlags = {restart_strategy(), MaxR, MaxT}
- restart_strategy() = one_for_all | one_for_one | rest_for_one | simple_one_for_one
- MaxR = int() >= 0
- MaxT = int() > 0
- ChildSpec = child_spec()

This function returns a supervisor specification. `ChildSpec` is as previously defined in the `start_child/2` function. `MaxR` is the maximum number of restarts which can be performed within `MaxT` seconds.

When the restart strategy is `simple_one_for_one`, the list of child specifications must be a list with one element only. This child is not started during the initialization phase, but all children are started dynamically. Each dynamically started child is of the same type, which means that all children are instances of the initial child specification. New children are created with a call to `start_child(Supervisor, ExtraStartArgs)`.

If a child start function returns `ignore`, the child is kept in the supervisor's list of children. The child can be restarted explicitly by calling `restart_child/2`. The child is also restarted if the supervisor is `one_for_all` and performs a restart of all children, or if the supervisor is `rest_for_one` and performs a restart of this child. The supervisor start-up fails and terminates if the child start function returns `{error, Reason}`

This function can return `ignore` in order to inform the parent, especially if it is another supervisor, that the supervisor is not started according to configuration data, for instance.

System Events

The supervisor behaviour generates the same system events as the `gen_server` behaviour. System events are handled by the `sys` module.

See Also

gen_server(3), sys(3)

supervisor_bridge (Module)

It can sometimes be useful to connect a process or a sub-system, which has not been designed with the supervision principles in mind, to a supervisor tree. This can be accomplished by using an instance of the `supervisor_bridge` behaviour. A supervisor bridge is a process which sits in between a supervisor and the sub-system. It behaves like a real supervisor to its own supervisor, but has a different interface than a real supervisor to the sub-system. Note, however, that it does not allow the use of the sophisticated code changing mechanisms to the sub-system.

An instance of the `supervisor_bridge` behaviour can be debugged with the module `sys`.

In the following, `Module` is the name of the callback module that implements the supervisor bridge behaviour.

Exports

```
start_link(Module,StartArgs) -> {ok, Pid} | ignore | {error, Reason}
start_link(Name,Module,StartArgs) -> {ok, Pid} | ignore | {error, Reason}
```

Types:

- Name = {local, atom()} | {global, atom()}
- Module = atom()
- StartArgs = term()

Starts a new supervisor bridge process synchronously. The function `Module:init(StartArgs)` is called (see below).

If the supervisor bridge is started with `Name`, the name is registered locally or globally.

Callback Functions

The following functions should be exported from a `supervisor_bridge` callback module.

Exports

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

Types:

- StartArgs = term()
- State = term()

This function starts the sub-system and returns the `Pid` of the main process in the sub-system, and a `State`. The `State` can be any term and it is sent to the `Module:terminate/2` function (see below).

`Module:terminate(Reason, State) -> void()`

Types:

- Reason = term()
- State = term()

This function terminates the sub-system. The return value is ignored.

System Events

The `supervisor_bridge` behaviour generates the same system events as the `gen_server` behaviour. System events are handled by the `sys` module.

See Also

gen_server(3), supervisor(3), sys(3)

sys (Module)

This module contains functions for sending system messages used by programs, and messaged used for debugging purposes.

Functions used for implementation of processes should also understand system messages such as debugging messages and code change. These functions must be used to implement the use of system messages for a process; either directly, or through standard behaviours, such as `gen_server`.

The following types are used in the functions defined below:

- `Name = pid() | atom() | {global, atom()}`
- `Timeout = int() >= 0 | infinity`
- `system_event() = {in, Msg} | {in, Msg, From} | {out, Msg, To} | term()`

The default timeout is 5000 ms, unless otherwise specified. The `timeout` defines the time period to wait for the process to respond to a request. If the process does not respond, the function evaluates `exit({timeout, {M, F, A}})`.

The functions make reference to a debug structure. The debug structure is a list of `dbg_opt()`. `dbg_opt()` is an internal data type used by the `handle_system_msg/6` function. No debugging is performed if it is an empty list.

System Messages

Processes which are not implemented as one of the standard behaviours must still understand system messages. There are three different messages which must be understood:

- Plain system messages. These are received as `{system, From, Msg}`. The content and meaning of this message are not interpreted by the receiving process module. When a system message has been received, the function `sys:handle_system_msg/6` is called in order to handle the request.
- Shutdown messages. If the process traps exits, it must be able to handle an shut-down request from its parent, the supervisor. The message `{'EXIT', Parent, Reason}` from the parent is an order to terminate. The process must terminate when this message is received, normally with the same `Reason` as `Parent`.

- There is one more message which the process must understand if the modules used to implement the process change dynamically during runtime. An example of such a process is the `gen_event` processes. This message is `{get_modules, From}`. The reply to this message is `From ! {modules, Modules}`, where `Modules` is a list of the currently active modules in the process.

  This message is used by the release handler to find which processes execute a certain module. The process may at a later time be suspended and ordered to perform a code change for one of its modules.

System Events

When debugging a process with the functions of this module, the process generates *system_events* which are then treated in the debug function. For example, `trace` formats the system events to the tty.

There are three predefined system events which are used when a process receives or sends a message. The process can also define its own system events. It is always up to the process itself to format these events.

Exports

`log(Name,Flag)`
`log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}`

Types:

- Flag = true | {true, N} | false | get | print
- N = integer() > 0

Turns the logging of system events On or Off. If On, a maximum of `N` events are kept in the debug structure (the default is 10). If `Flag` is `get`, a list of all logged events is returned. If `Flag` is `print`, the logged events are printed to `standard_io`. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`log_to_file(Name,Flag)`
`log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`

Types:

- Flag = FileName | false
- FileName = string()

Enables or disables the logging of all system events in textual format to the file. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`statistics(Name,Flag)`
`statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`

Types:

- Flag = true | false | get
- Statistics = [{start_time, {Date1, Time1}}, {current_time, {Date, Time2}}, {reductions, integer()}, {messages_in, integer()}, {messages_out, integer()}]
- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Min, Sec}

Enables or disables the collection of statistics. If `Flag` is `get`, the statistical collection is returned.

`trace(Name,Flag)`
`trace(Name,Flag,Timeout) -> void()`

Types:

- Flag = boolean()

Prints all system events on `standard_io`. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`no_debug(Name)`
`no_debug(Name,Timeout) -> void()`

Turns off all debugging for the process. This includes functions that have been installed explicitly with the `install` function, for example triggers.

`suspend(Name)`
`suspend(Name,Timeout) -> void()`

Suspends the process. When the process is suspended, it will only respond to other system messages, but not other messages.

`resume(Name)`
`resume(Name,Timeout) -> void()`

Resumes a suspended process.

`change_code(Name, OldVsn, Module, Extra)`
`change_code(Name, OldVsn, Module, Extra, Timeout) -> ok | {error, Reason}`

Types:

- OldVsn = undefined | term()
- Module = atom()
- Extra = term()

Tells the process to change code. The process must be suspended to handle this message. The `Extra` argument is reserved for each process to use as its own. The function `Mod:system_code_change/4` is called. `OldVsn` is the old version of the `Module`.

`get_status(Name)`
`get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`

Types:

- PDict = [{Key, Value}]
- SysState = running | suspended
- Parent = pid()
- Dbg = [dbg_opt()]
- Misc = term()

Gets the status of the process.

`install(Name,{Func,FuncState})`
`install(Name,{Func,FuncState},Timeout)`

Types:

- Func = dbg_fun()
- dbg_fun() = fun(FuncState, Event, ProcState) -> done | NewFuncState
- FuncState = term()
- Event = system_event()
- ProcState = term()
- NewFuncState = term()

This function makes it possible to install other debug functions than the ones defined above. An example of such a function is a trigger, a function that waits for some special event and performs some action when the event is generated. This could, for example, be turning on low level tracing.

`Func` is called whenever a system event is generated. This function should return `done`, or a new func state. In the first case, the function is removed. It is removed if the function fails.

`remove(Name,Func)`
`remove(Name,Func,Timeout) -> void()`

Types:

- Func = dbg_fun()

Removes a previously installed debug function from the process. `Func` must be the same as previously installed.

Process Implementation Functions

The following functions are used when implementing a special process. This is an ordinary process which does not use a standard behaviour, but a process which understands the standard system messages.

Exports

`debug_options(Options) -> [dbg_opt()]`

Types:

- Options = [Opt]
- Opt = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
- Func = dbg_fun()
- FuncState = term()

This function can be used by a process that initiates a debug structure from a list of options. The values of the `Opt` argument are the same as the corresponding functions.

`get_debug(Item,Debug,Default) -> term()`

Types:

- Item = log | statistics
- Debug = [dbg_opt()]
- Default = term()

This function gets the data associated with a debug option. `Default` is returned if the `Item` is not found. Can be used by the process to retrieve debug data for printing before it terminates.

`handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]`

Types:

- FormFunc = dbg_fun()
- Extra = term()
- Event = system_event()

This function is called by a process when it generates a system event. `FormFunc` is a formatting function which is called as `FormFunc(Device, Event, Extra)` in order to print the events, which is necessary if tracing is activated. `Extra` is any extra information which the process needs in the format function, for example the name of the process.

`handle_system_msg(Msg,From,Parent,Module,Debug,Misc)`

Types:

- Msg = term()
- From = pid()
- Parent = pid()
- Module = atom()
- Debug = [dbg_opt()]
- Misc = term()

This function is used by a process module that wishes to take care of system messages. The process receives a {`system, From, Msg`} message and passes the `Msg` and `From` to this function.

This function *never* returns. It calls the function `Module:system_continue(Parent, NDebug, Misc)` where the process continues the execution, or `Module:system_terminate(Reason, Parent, Debug, Misc)` if the process should terminate. The `Module` must export `system_continue/3`, `system_terminate/4`, and `system_code_change/4` (see below).

The `Misc` argument can be used to save internal data in a process, for example its state. It is sent to `Module:system_continue/3` or `Module:system_terminate/4`

`print_log(Debug) -> void()`

Types:

- Debug = [dbg_opt()]

Prints the logged system events in the debug structure using `FormFunc` as defined when the event was generated by a call to `handle_debug/4`.

`Mod:system_continue(Parent, Debug, Misc)`

Types:

- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from `sys:handle_system_msg/6` when the process should continue its execution (for example after it has been suspended). This function never returns.

`Mod:system_terminate(Reason, Parent, Debug, Misc)`

Types:

- Reason = term()
- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from `sys:handle_system_msg/6` when the process should terminate. For example, this function is called when the process is suspended and its parent orders shut-down. It gives the process a chance to do a clean-up. This function never returns.

`Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}`

Types:

- Misc = term()
- OldVsn = undefined | term()
- Module = atom()
- Extra = term()
- NMisc = term()

Called from `sys:handle_system_msg/6` when the process should perform a code change. The code change is used when the internal data structure has changed. This function converts the `Misc` argument to the new data structure. `OldVsn` is the *vsn* attribute of the old version of the `Module`. If no such attribute was defined, the atom `undefined` is sent.

timer (Module)

This module provides useful functions related to time. Unless otherwise stated, time is always measured in `milliseconds`. All timer functions return immediately, regardless of work carried out by another process.

Successful evalutions of the timer functions yield return values containing a timer reference, denoted `TRef` below. By using `cancel/1`, the returned reference can be used to cancel any requested action. A `TRef` is an Erlang term, the contents of which must not be altered.

The timeouts are not exact, but should be `at least` as long as requested.

Exports

`start() -> ok`

Starts the timer server. Normally, the server does not need to be started explicitly. 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.

`apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}`

Types:

- Time = integer() in Milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates `apply(M, F, A)` after `Time` amount of time has elapsed. Returns `{ok, TRef}`, or `{error, Reason}`.

`send_after(Time, Pid, Message) -> {ok, TRef} | {error,Reason}`
`send_after(Time, Message) -> {ok, TRef} | {error,Reason}`

Types:

- Time = integer() in Milliseconds
- Pid = pid() | atom()
- Message = term()
- Result = {ok, TRef} | {error, Reason}

`send_after/3` Evaluates `Pid ! Message` after `Time` amount of time has elapsed. (`Pid` can also be an atom of a registered name.) Returns `{ok, TRef}`, or `{error, Reason}`.

`send_after/2` Same as `send_after(Time, self(), Message)`.

`exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}`
`exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}`
`kill_after(Time, Pid)-> {ok, TRef} | {error,Reason2}`
`kill_after(Time) -> {ok, TRef} | {error,Reason2}`

Types:

- Time = integer() in milliseconds
- Pid = pid() | atom()
- Reason1 = Reason2 = term()

`exit_after/3` Send an exit signal with reason `Reason1` to Pid `Pid`. Returns `{ok, TRef}`, or `{error, Reason2}`.

`exit_after/2` Same as `exit_after(Time, self(), Reason1)`.

`kill_after/2` Same as `exit_after(Time, Pid, kill)`.

`kill_after/1` Same as `exit_after(Time, self(), kill)`.

`apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`

Types:

- Time = integer() in milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates `apply(Module, Function, Arguments)` repeatedly at intervals of `Time`. Returns `{ok, TRef}`, or `{error, Reason}`.

`send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
`send_interval(Time, Message) -> {ok, TRef} | {error, Reason}`

Types:

- Time = integer() in milliseconds
- Pid = pid() | atom()
- Message = term()
- Reason = term()

`send_interval/3` Evaluates `Pid ! Message` repeatedly after `Time` amount of time has elapsed. (`Pid` can also be an atom of a registered name.) Returns `{ok, TRef}` or `{error, Reason}`.

`send_interval/2` Same as `send_interval(Time, self(), Message)`.

`cancel(TRef) -> {ok, cancel} | {error, Reason}`

Cancels a previously requested timeout. `TRef` is a unique timer reference returned by the timer function in question. Returns `{ok, cancel}`, or `{error, Reason}` when `TRef` is not a timer reference.

`sleep(Time) -> ok`

Types:

- Time = integer() in milliseconds

Suspends the process calling this function for `Time` amount of milliseconds and then returns `ok`. Naturally, this function does *not* return immediately.

`tc(Module, Function, Arguments) -> {Time, Value}`

Types:

- Module = Function = atom()
- Arguments = [term()]
- Time = integer() in microseconds
- Value = term()

Evaluates `apply(Module, Function, Arguments)` and measures the elapsed real time. Returns `{Time, Value}`, where `Time` is the elapsed real time in *microseconds*, and `Value` is what is returned from the apply.

`seconds(Seconds) -> Milliseconds`

Returns the number of milliseconds in `Seconds`.

`minutes(Minutes) -> Milliseconds`

Returns the number of milliseconds in `Minutes`.

`hours(Hours) -> Milliseconds`

Returns the number of milliseconds in `Hours`.

`hms(Hours, Minutes, Seconds) -> Milliseconds`

Returns the number of milliseconds in `Hours + Minutes + Seconds`.

Examples

This example illustrates how to print out “Hello World!” in 5 seconds:

```
1> timer:apply_after(5000, io, format, ["~nHello World!~n", []]).
{ok,TRef}
Hello World!
2>
```

The following coding example illustrates a process which performs a certain action and if this action is not completed within a certain limit, then the process is killed.

```
Pid = spawn(mod, fun, [foo, bar]),
%% If pid is not finished in 10 seconds, kill him
{ok, R} = timer:kill_after(timer:seconds(10), Pid),
...
%% We change our mind...
timer:cancel(R),
...
```

WARNING

A timer can always be removed by calling `cancel/1`.

An interval timer, i.e. a timer created by evaluating any of the functions `apply_interval/4`, `send_interval/3`, and `send_interval/2`, is linked to the process towards which the timer performs its task.

A one-shot timer, i.e. a timer created by evaluating any of the functions `apply_after/4`, `send_after/3`, `send_after/2`, `exit_after/3`, `exit_after/2`, `kill_after/2`, and `kill_after/1` is not linked to any process. Hence, such a timer is removed only when it reaches its timeout, or if it is explicitely removed by a call to `cancel/1`.

unix (Module)

This module makes it possible to make calls to the UNIX shell. The shell used is `/bin/sh`, so the environment might be different to the one you commonly use. C shell expansions cannot be used. The module is extremely easy to use and there is only one function.

Note that most UNIX commands produce a trailing new line.

Exports

`cmd(String)`

Makes the call `String` to `sh` and returns the answer in a list of characters.

Example: (bizarre version of ls)

```
1> unix:cmd("for i in *; do echo $i; done").
```

Index

Modules are typed in *this way*.
Functions are typed in `this way`.