STDLIB

version 1.12

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

Contents

STDLIB Reference Manual

Short Summaries

beam_lib

The following functions are exported:

c

The following functions are exported:

- `ls() -> void()`
  [page 53] List files
- `ls(Dir) -> void()`
  [page 53] List files in `Dir`
- `m() -> void()`
  [page 53] List all loaded modules
- `m(Module) -> void()`
  [page 54] Display information about a module
- `memory() -> MemoryInformation`
  [page 54] Returns memory information on dynamically allocated memory
- `memory(MemoryTypeSpecification) -> MemoryInformation`
  [page 54] Returns memory information on dynamically allocated memory
- `nc(File) -> void()`
  [page 54] Compile file and loads it on multiple nodes
- `nc(File, Flags) -> void()`
  [page 54] Compile file and loads it on multiples nodes
- `ni() -> void()`
  [page 54] Display network information
- `nl(Module) -> void()`
  [page 54] Load module in a network
- `nregs() -> void()`
  [page 54] Display registered processes on all nodes
- `pid(X, Y, Z) -> pid()`
  [page 54] Make a Pid
- `pwd() -> void()`
  [page 55] Print current working directory
- `q() -> void()`
  [page 55] Stop the Erlang node
- `regs() -> void()`
  [page 55] Display registered processes
- `xm(ModSpec) -> void()`
  [page 55] Cross reference check a module
- `zi() -> void()`
  [page 55] Display system information including zombies

calendar

The following functions are exported:

- `date_to_gregorian_days(Year, Month, Day) -> Days`
  [page 56] Compute the number of days from year 0 up to the given date.
- `date_to_gregorian_days(Date) -> Days`
  [page 56] Compute the number of days from year 0 up to the given date.
- `datetime_to_gregorian_seconds(DateTime) -> Days`
  [page 56] Compute the number of seconds from year 0 up to the given date and time.

- `day_of_the_week(Date) -> DayNumber`
  [page 57] Compute the day of the week
- `day_of_the_week(Year, Month, Day) -> DayNumber`
  [page 57] Compute the day of the week
- `gregorian_days_to_date(Days) -> Date`
  [page 57] Compute the date given the number of gregorian days.
- `gregorian_seconds_to_datetime(Secs) -> DateTime`
  [page 57] Compute the date given the number of gregorian days.
- `is_leap_year(Year) -> bool()`
  [page 57] Check if a year is a leap year.
- `last_day_of_the_month(Year, Month) -> int()`
  [page 57] Compute the number of days in a month
- `local_time() -> {Date, Time}`
  [page 58] Compute local time
- `local_time_to_universal_time({Date, Time}) -> {Date, Time}`
  [page 58] Convert from local time to universal time.
- `local_time_to_universal_time_dst(DateTime) -> [DateTimeUTC]`
  [page 58] Convert from local time to universal time(s).
- `now_to_local_time(Now) -> {Date, Time}`
  [page 58] Convert now to local date and time
- `now_to_universal_time(Now) -> {Date, Time}`
  [page 59] Convert now to date and time
- `now_to_datetime(Now) -> {Date, Time}`
  [page 59] Convert now to date and time
- `seconds_to_daystime(Secs) -> {Days, Time}`
  [page 59] Compute a days and time from seconds.
- `seconds_to_time(Secs) -> Time`
  [page 59] Compute time from seconds.
- `time_difference(T1, T2) -> Tdiff`
  [page 59] Compute the difference between two times
- `time_to_secnds(Time) -> Secs`
  [page 60] Compute the number of seconds since midnight up to the given time.
- `universal_time() -> {Date, Time}`
  [page 60] Compute universal time
- `universal_time_to_local_time({Date, Time}) -> {Date, Time}`
  [page 60] Convert from universal time to local time.
- `valid_date(Date) -> bool()`
  [page 60] Check if a date is valid
- `valid_date(Year, Month, Day) -> bool()`
  [page 60] Check if a date is valid

dets

The following functions are exported:

- `all() -> [Name]`
  [page 63] Return a list of the names of all open Dets tables on this node.

dict

The following functions are exported:

- `append(Key, Value, Dict1) -> Dict2`
  [page 77] Append a value to keys in a dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`
  [page 77] Append new values to keys in a dictionary
- `erase(Key, Dict1) -> Dict2`
  [page 77] Erase a key from a dictionary
- `fetch(Key, Dict) -> Value`
  [page 77] Look-up values in a dictionary
- `fetch_keys(Dict) -> Keys`
  [page 78] Return all keys in a dictionary
- `filter(Pred, Dict1) -> Dict2`
  [page 78] Choose elements which satisfy a predicate
- `find(Key, Dict) -> Result`
  [page 78] Searche for a key in a dictionary
- `fold(Function, Acc0, Dict) -> Acc1`
  [page 78] Fold a function over a dictionary
- `from_list(List) -> Dict`
  [page 78] Convert a list of pairs to a dictionary
- `is_key(Key, Dict) -> bool()`
  [page 78] Test if a key is in a dictionary.
- `map(Func, Dict1) -> Dict2`
  [page 78] Map a function over a dictionary
- `merge(Func, Dict1, Dict2) -> Dict3`
  [page 79] Merge two dictionaries
- `new() -> dictionary()`
  [page 79] Create a dictionary
- `store(Key, Value, Dict1) -> Dict2`
  [page 79] Store a value in a dictionary
- `to_list(Dict) -> List`
  [page 79] Convert a dictionary to a list of pairs
- `update(Key, Function, Dict) -> Dict`
  [page 79] Update a value in a dictionary
- `update(Key, Function, Initial, Dict) -> Dict`
  [page 80] Update a value in a dictionary
- `update_counter(Key, Increment, Dict) -> Dict`
  [page 80] Increment a value in a dictionary

digraph

The following functions are exported:

- `add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}`
  [page 81] Add an edge to a digraph.

- add_edge(G, V1, V2, Label) -> edge() | {error, Reason}
  [page 81] Add an edge to a digraph.
- add_edge(G, V1, V2) -> edge() | {error, Reason}
  [page 81] Add an edge to a digraph.
- add_vertex(G, V, Label) -> vertex()
  [page 82] Add or modify a vertex of a digraph.
- add_vertex(G, V) -> vertex()
  [page 82] Add or modify a vertex of a digraph.
- add_vertex(G) -> vertex()
  [page 82] Add or modify a vertex of a digraph.
- del_edge(G, E) -> true
  [page 82] Delete an edge from a digraph.
- del_edges(G, Edges) -> true
  [page 82] Delete edges from a digraph.
- del_path(G, V1, V2) -> true
  [page 82] Delete paths from a digraph.
- del_vertex(G, V) -> true
  [page 83] Delete a vertex from a digraph.
- del_vertices(G, Vertices) -> true
  [page 83] Delete vertices from a digraph.
- delete(G) -> true
  [page 83] Delete a digraph.
- edge(G, E) -> {E, V1, V2, Label} | false
  [page 83] Return the vertices and the label of an edge of a digraph.
- edges(G) -> Edges
  [page 83] Return all edges of a digraph.
- edges(G, V) -> Edges
  [page 83] Return the edges emanating from or incident on a vertex of a digraph.
- get_cycle(G, V) -> Vertices | false
  [page 84] Find one cycle in a digraph.
- get_path(G, V1, V2) -> Vertices | false
  [page 84] Find one path in a digraph.
- get_short_cycle(G, V) -> Vertices | false
  [page 84] Find one short cycle in a digraph.
- get_short_path(G, V1, V2) -> Vertices | false
  [page 84] Find one short path in a digraph.
- in_degree(G, V) -> integer()
  [page 85] Return the in-degree of a vertex of a digraph.
- in_edges(G, V) -> Edges
  [page 85] Return all edges incident on a vertex of a digraph.
- in_neighbours(G, V) -> Vertices
  [page 85] Return all in-neighbours of a vertex of a digraph.
- info(G) -> InfoList
  [page 85] Return information about a digraph.
- new() -> digraph()
  [page 86] Return a protected empty digraph, where cycles are allowed.

- `new(Type) -> digraph() | {error, Reason}`
  [page 86] Create a new empty digraph.
- `no_edges(G) -> integer() >= 0`
  [page 86] Return the number of edges of the a digraph.
- `no_vertices(G) -> integer() >= 0`
  [page 86] Return the number of vertices of a digraph.
- `out_degree(G, V) -> integer()`
  [page 86] Return the out-degree of a vertex of a digraph.
- `out_edges(G, V) -> Edges`
  [page 86] Return all edges emanating from a vertex of a digraph.
- `out_neighbours(G, V) -> Vertices`
  [page 86] Return all out-neighbours of a vertex of a digraph.
- `vertex(G, V) -> {V, Label} | false`
  [page 87] Return the label of a vertex of a digraph.
- `vertices(G) -> Vertices`
  [page 87] Return all vertices of a digraph.

digraph_utils

The following functions are exported:

- `components(Digraph) -> [Component]`
  [page 89] Return the components of a digraph.
- `condensation(Digraph) -> CondensedDigraph`
  [page 89] Return a condensed graph of a digraph.
- `cyclic_strong_components(Digraph) -> [StrongComponent]`
  [page 89] Return the cyclic strong components of a digraph.
- `is_acyclic(Digraph) -> bool()`
  [page 89] Check if a digraph is acyclic.
- `loop_vertices(Digraph) -> Vertices`
  [page 89] Return the vertices of a digraph included in some loop.
- `postorder(Digraph) -> Vertices`
  [page 90] Return the vertices of a digraph in post-order.
- `preorder(Digraph) -> Vertices`
  [page 90] Return the vertices of a digraph in pre-order.
- `reachable(Vertices, Digraph) -> Vertices`
  [page 90] Return the vertices reachable from some vertices of a digraph.
- `reachable_neighbours(Vertices, Digraph) -> Vertices`
  [page 90] Return the neighbours reachable from some vertices of a digraph.
- `reaching(Vertices, Digraph) -> Vertices`
  [page 90] Return the vertices that reach some vertices of a digraph.
- `reaching_neighbours(Vertices, Digraph) -> Vertices`
  [page 90] Return the neighbours that reach some vertices of a digraph.
- `strong_components(Digraph) -> [StrongComponent]`
  [page 91] Return the strong components of a digraph.

- subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}
  [page 91] Return a subgraph of a digraph.
- topsort(Digraph) -> Vertices | false
  [page 91] Return a topological sorting of the vertices of a digraph.

epp

The following functions are exported:

- open(FileName, IncludePath) -> {ok,Epp} | {error, ErrorDescriptor}
  [page 92] Open a file for preprocessing
- open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}
  [page 92] Open a file for preprocessing
- close(Epp) -> ok
  [page 92] Close the preprocessing of the file associated with Epp
- parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}
  [page 92] Return the next Erlang form from the opened Erlang source file
- parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}
  [page 92] Preprocesse and parse an Erlang source file

erl_eval

The following functions are exported:

- exprs(Expressions, Bindings) -> {value, Value, NewBindings}
  [page 94] Evaluate expressions
- exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}
  [page 94] Evaluate expressions
- exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}
  [page 94] Evaluate expressions
- expr(Expression, Bindings) -> { value, Value, NewBindings }
  [page 94] Evaluate expression
- expr(Expression, Bindings, LocalFunctionHandler) -> { value, Value, NewBindings }
  [page 94] Evaluate expression
- expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> { value, Value, NewBindings }
  [page 94] Evaluate expression
- expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}
  [page 95] Evaluate a list of expressions
- expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}
  [page 95] Evaluate a list of expressions

- `expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 95] Evaluate a list of expressions
- `new_bindings() -> BindingStruct`
  [page 95] Return a bindings structure
- `bindings(BindingStruct) -> Bindings`
  [page 95] Return bindings
- `binding(Name, BindingStruct) -> Binding`
  [page 95] Return bindings
- `add_binding(Name, Value, Bindings) -> BindingStruct`
  [page 95] Add a binding
- `del_binding(Name, Bindings) -> BindingStruct`
  [page 95] Delete a binding

erl_id_trans

The following functions are exported:

- `parse_transform(Forms, Options) -> Forms`
  [page 97] Transform Erlang forms

erl_internal

The following functions are exported:

- `bif(Name, Arity) -> bool()`
  [page 98] Test for an Erlang BIF
- `guard_bif(Name, Arity) -> bool()`
  [page 98] Test for an Erlang BIF allowed in guards
- `type_test(Name, Arity) -> bool()`
  [page 98] Test for a valid type test
- `arith_op(OpName, Arity) -> bool()`
  [page 98] Test for an arithmetic operator
- `bool_op(OpName, Arity) -> bool()`
  [page 98] Test for a Boolean operator
- `comp_op(OpName, Arity) -> bool()`
  [page 99] Test for a comparison operator
- `list_op(OpName, Arity) -> bool()`
  [page 99] Test for a list operator
- `send_op(OpName, Arity) -> bool()`
  [page 99] Test for a send operator
- `op_type(OpName, Arity) -> Type`
  [page 99] Return operator type

erl_lint

The following functions are exported:

- `module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 100] Check a module for errors
- `module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 100] Check a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 100] Check a module for errors
- `is_guard_test(Expr) -> bool()`
  [page 101] Test for a guard test
- `format_error(ErrorDescriptor) -> string()`
  [page 101] Format an error descriptor

erl_parse

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`
  [page 103] Parse an Erlang form
- `parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`
  [page 103] Parse Erlang expressions
- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`
  [page 103] Parse an Erlang term
- `format_error(ErrorDescriptor) -> string()`
  [page 104] Format an error descriptor
- `tokens(AbsTerm) -> Tokens`
  [page 104] Generate a list of tokens for an expression
- `tokens(AbsTerm, MoreTokens) -> Tokens`
  [page 104] Generate a list of tokens for an expression
- `normalise(AbsTerm) -> Data`
  [page 104] Convert abstract form to an Erlang term
- `abstract(Data) -> AbsTerm`
  [page 104] Convert an Erlang term into an abstract form

erl_pp

The following functions are exported:

- `form(Form) -> DeepCharList`
  [page 106] Pretty print a form
- `form(Form, HookFunction) -> DeepCharList`
  [page 106] Pretty print a form
- `attribute(Attribute) -> DeepCharList`
  [page 106] Pretty print an attribute

- `attribute(Attribute, HookFunction) -> DeepCharList`
  [page 106] Pretty print an attribute
- `function(Function) -> DeepCharList`
  [page 106] Pretty print a function
- `function(Function, HookFunction) -> DeepCharList`
  [page 106] Pretty print a function
- `guard(Guard) -> DeepCharList`
  [page 106] Pretty print a guard
- `guard(Guard, HookFunction) -> DeepCharList`
  [page 106] Pretty print a guard
- `exprs(Expressions) -> DeepCharList`
  [page 107] Pretty print `Expressions`
- `exprs(Expressions, HookFunction) -> DeepCharList`
  [page 107] Pretty print `Expressions`
- `exprs(Expressions, Indent, HookFunction) -> DeepCharList`
  [page 107] Pretty print `Expressions`
- `expr(Expression) -> DeepCharList`
  [page 107] Pretty print one `Expression`
- `expr(Expression, HookFunction) -> DeepCharList`
  [page 107] Pretty print one `Expression`
- `expr(Expression, Indent, HookFunction) -> DeepCharList`
  [page 107] Pretty print one `Expression`
- `expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList`
  [page 107] Pretty print one `Expression`

erl_scan

The following functions are exported:

- `string(CharList,StartLine]) -> {ok, Tokens, EndLine} | Error`
  [page 109] Scan a string and returns the Erlang tokens
- `string(CharList) -> {ok, Tokens, EndLine} | Error`
  [page 109] Scan a string and returns the Erlang tokens
- `tokens(Continuation, CharList, StartLine) ->Return`
  [page 109] Re-entrant scanner
- `reserved_word(Atom) -> bool()`
  [page 110] Test for a reserved word
- `format_error(ErrorDescriptor) -> string()`
  [page 110] Format an error descriptor

erl_tar

The following functions are exported:

- `add(TarDescriptor, Filename, Options) -> RetValue`
  [page 112] Add a file to an open tar file

ets

The following functions are exported:

- `member(Tab, Key) -> true | false`
  [page 126] Tests for occurrence of a key in an ETS table
- `new(Name, Options) -> tid()`
  [page 126] Create a new ETS table.
- `next(Tab, Key1) -> Key2 | '$end_of_table'`
  [page 127] Return the next key in an ETS table.
- `prev(Tab, Key1) -> Key2 | '$end_of_table'`
  [page 127] Return the previous key in an ETS table of type `ordered_set`.
- `rename(Tab, Name) -> Name`
  [page 127] Rename a named ETS table.
- `safe_fixtable(Tab, true|false) -> true | false`
  [page 128] Fix an ETS table for safe traversal.
- `select(Tab, MatchSpec) -> [Object]`
  [page 128] Match the objects in an ETS table against a match_spec.
- `select(Tab, MatchSpec, Limit) -> {[Match],Continuation} | '$end_of_table'`
  [page 130] Match the objects in an ETS table against a match_spec and returns part of the answers.
- `select(Continuation) -> {[Match],Continuation} | '$end_of_table'`
  [page 130] Continues matching objects in an ETS table.
- `select_delete(Tab, MatchSpec) -> NumDeleted`
  [page 130] Match the objects in an ETS table against a match_spec and deletes objects where the match_spec returns 'true'
- `select_count(Tab, MatchSpec) -> NumMatched`
  [page 131] Match the objects in an ETS table against a match_spec and returns the number of objects for which the match_spec returned 'true'
- `slot(Tab, I) -> [Object] | '$end_of_table'`
  [page 131] Return all objects in a given slot of an ETS table.
- `tab2file(Tab, Filename) -> ok | {error,Reason}`
  [page 131] Dump an ETS table to a file.
- `tab2list(Tab) -> [Object]`
  [page 132] Return a list of all objects in an ETS table.
- `test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`
  [page 132] Test a match_spec for use in ets:select/2.
- `to_dets(Tab, DetsTab) -> Tab`
  [page 132] Fill a DETS table withe objects from an ETS table.
- `update_counter(Tab, Key, {Pos,Incr,Threshold,SetValue}) -> Result`
  [page 132] Update a counter object in an ETS table.
- `update_counter(Tab, Key, {Pos,Incr}) -> Result`
  [page 132] Update a counter object in an ETS table.
- `update_counter(Tab, Key, Incr) -> Result`
  [page 132] Update a counter object in an ETS table.

file_sorter

The following functions are exported:

- `sort(FileName) -> Reply`
  [page 137] Sort terms on files.
- `sort(Input, Output) -> Reply`
  [page 137] Sort terms on files.
- `sort(Input, Output, Options) -> Reply`
  [page 137] Sort terms on files.
- `keysort(KeyPos, FileName) -> Reply`
  [page 137] Sort terms on files by key.
- `keysort(KeyPos, Input, Output) -> Reply`
  [page 137] Sort terms on files by key.
- `keysort(KeyPos, Input, Output, Options) -> Reply`
  [page 137] Sort terms on files by key.
- `merge(FileNames, Output) -> Reply`
  [page 137] Merge terms on files.
- `merge(FileNames, Output, Options) -> Reply`
  [page 137] Merge terms on files.
- `keymerge(KeyPos, FileNames, Output) -> Reply`
  [page 138] Merge terms on files by key.
- `keymerge(KeyPos, FileNames, Output, Options) -> Reply`
  [page 138] Merge terms on files by key.
- `check(FileName) -> Reply`
  [page 138] Check whether terms on files are sorted.
- `check(FileNames, Options) -> Reply`
  [page 138] Check whether terms on files are sorted.
- `keycheck(KeyPos, FileName) -> CheckReply`
  [page 138] Check whether terms on files are sorted by key.
- `keycheck(KeyPos, FileNames, Options) -> Reply`
  [page 138] Check whether terms on files are sorted by key.

filelib

The following functions are exported:

- `ensure_dir(Name) -> true`
  [page 139] Ensure that all parent directories needed to create `Name` exists.
- `file_size(Filename) -> integer()`
  [page 139] Return the size in bytes of the file.
- `fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`
  [page 139] Fold over all files matching a regular expression.
- `is_dir(Name) -> true | false`
  [page 139] Test whether Name refer to a directory or not
- `is_file(Name) -> true | false`
  [page 139] Test whether Name refer to a file or directory.
- `last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}}`
  [page 140] Return the local date and time when a file was last modified.
- `wildcard(Wildcard) -> list()`
  [page 140] Match filenames using Unix-style wildcards.

- `wildcard(Wildcard, Cwd) -> list()`
  [page 140] Match filenames using Unix-style wildcards startin at a specified directory.

filename

The following functions are exported:

- `absname(Filename) -> Absname`
  [page 142] Convert a relative `Filename` to an absolute name
- `absname(Filename, Directory) -> Absname`
  [page 142] Convert the relative `Filename` to an absolute name, based on `Directory`.
- `absname_join(Directory, Filename) -> Absname`
  [page 143] Join an absolute directory with a relative filename.
- `basename(Filename)`
  [page 143] Return the part of the `Filename` after the last directory separator
- `basename(Filename,Ext) -> string()`
  [page 143] Return the last component of `Filename` with `Ext`stripped
- `dirname(Filename) -> string()`
  [page 144] Return the directory part of a path name
- `extension(Filename) -> string() | []`
  [page 144] Return the file extension
- `flatten(Filename) -> string()`
  [page 144] Convert a filename to a flat string
- `join(Components) -> string()`
  [page 144] Join a list of file name `Components` with directory separators
- `join(Name1, Name2) -> string()`
  [page 144] Join two file name components with directory separators.
- `nativename(Path) -> string()`
  [page 145] Return the native form of a file Path
- `pathtype(Path) -> absolute | relative | volumerelative`
  [page 145] Return the type of a Path
- `rootname(Filename) -> string()`
  [page 145] Return all characters in `Filename`, except the extension.
- `rootname(Filename, Ext) -> string()`
  [page 145] Return all characters in `Filename`, except the extension.
- `split(Filename) -> Components`
  [page 146] Return a list whose elements are the file name components of `Filename`.
- `find_src(Module) -> {SourceFile, Options}`
  [page 146] Find the `Filename` and compilation options for a compiled `Module`.
- `find_src(Module, Rules) -> {SourceFile, Options}`
  [page 146] Find the `Filename` and compilation options for a compiled `Module`.

gb_sets

The following functions are exported:

- `empty()`
  [page 147] get empty set
- `is_empty(S)`
  [page 147] check if empty
- `size(S)`
  [page 147] get number of elements
- `singleton(X)`
  [page 147] new set with one element
- `is_member(X, S)`
  [page 147] check for member
- `insert(X, S)`
  [page 148] insert new element
- `add(X, S)`
  [page 148] add element
- `delete(X, S)`
  [page 148] delete element
- `delete_any(X, T)`
  [page 148] removes key if present
- `balance(S)`
  [page 148] rebalance tree representation
- `union(S1, S2)`
  [page 148] union of set
- `union(Ss)`
  [page 148] union of list of sets
- `intersection(S1, S2)`
  [page 148] intersection of sets
- `intersection(Ss)`
  [page 148] intersection of list of sets
- `difference(S1, S2)`
  [page 148] difference of sets
- `is_subset(S1, S2)`
  [page 149] check for subset
- `to_list(S)`
  [page 149] get list from set
- `from_list(List)`
  [page 149] make set from list
- `from_ordset(L)`
  [page 149] make set from ordset
- `smallest(S)`
  [page 149] return smallest element
- `largest(S)`
  [page 149] return largest element

gb_trees

The following functions are exported:

- `to_list(T)`
  [page 152] keys and values as tuple-list
- `from_orddict(L)`
  [page 152] make tree from orddict
- `smallest(T)`
  [page 153] return smallest key and value
- `largest(T)`
  [page 153] return smallest key and value
- `take_smallest(T)`
  [page 153] extract smallest key and value
- `take_largest(T)`
  [page 153] extract smallest key and value
- `iterator(T)`
  [page 153] get iterator on tree
- `next(S)`
  [page 153] iterate using iterator

gen_event

The following functions are exported:

- `start() -> Result`
  [page 155] Create a generic event manager.
- `start(EventMgrName) -> Result`
  [page 155] Create a generic event manager.
- `start_link() -> Result`
  [page 155] Create a generic event manager.
- `start_link(EventMgrName) -> Result`
  [page 155] Create a generic event manager.
- `add_handler(EventMgrRef, Handler, Args) -> Result`
  [page 155] Add an event handler to a generic event manager.
- `add_sup_handler(EventMgrRef, Handler, Args) -> Result`
  [page 156] Add a supervised event handler to a generic event manager.
- `notify(EventMgrRef, Event) -> ok`
  [page 156] Notify an event manager about an event.
- `sync_notify(EventMgrRef, Event) -> ok`
  [page 156] Notify an event manager about an event.
- `call(EventMgrRef, Handler, Request) -> Result`
  [page 157] Make a synchronous call to a generic event manager.
- `call(EventMgrRef, Handler, Request, Timeout) -> Result`
  [page 157] Make a synchronous call to a generic event manager.
- `delete_handler(EventMgrRef, Handler, Args) -> Result`
  [page 157] Delete an event handler from a generic event manager.
- `swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 158] Replace an event handler in a generic event manager.

- `swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 159] Replace an event handler in a generic event manager.
- `which_handlers(EventMgrRef) -> [Handler]`
  [page 159] Return all event handlers installed in a generic event manager.
- `stop(EventMgrRef) -> ok`
  [page 159] Terminate a generic event manager.
- `Module:init(InitArgs) -> {ok,State}`
  [page 160] Initialize an event handler.
- `Module:handle_event(Event, State) -> Result`
  [page 160] Handle an event.
- `Module:handle_call(Request, State) -> Result`
  [page 161] Handle a synchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 161] Handle an incoming message.
- `Module:terminate(Arg, State) -> term()`
  [page 161] Clean up before deletion.
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 162] Update the internal state due to code replacement

gen_fsm

The following functions are exported:

- `start(Module, Args, Options) -> Result`
  [page 164] Create a generic FSM process.
- `start(FsmName, Module, Args, Options) -> Result`
  [page 164] Create a generic FSM process.
- `start_link(Module, Args, Options) -> Result`
  [page 164] Create a generic FSM process.
- `start_link(FsmName, Module, Args, Options) -> Result`
  [page 164] Create a generic FSM process.
- `send_event(FsmRef, Event) -> ok`
  [page 165] Send an event asynchronously to a generic FSM.
- `send_all_state_event(FsmRef, Event) -> ok`
  [page 165] Send an event asynchronously to a generic FSM.
- `sync_send_event(FsmRef, Event) -> Reply`
  [page 165] Send an event synchronously to a generic FSM.
- `sync_send_event(FsmRef, Event, Timeout) -> Reply`
  [page 165] Send an event synchronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event) -> Reply`
  [page 166] Send an event syncronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply`
  [page 166] Send an event syncronously to a generic FSM.
- `reply(Caller, Reply) -> true`
  [page 166] Send a reply to a caller.

- `send_event_after(Time, Event) -> Ref`
  [page 166] Send a delayed event internally in a generic FSM.
- `start_timer(Time, Msg) -> Ref`
  [page 167] Send a timeout event internally in a generic FSM.
- `cancel_timer(Ref) -> RemainingTime | false`
  [page 167] Cancel an internal timer in a generic FSM.
- `Module:init(Args) -> Result`
  [page 168] Initialize process and internal state name and state data.
- `Module:StateName(Event, StateData) -> Result`
  [page 168] Handle an asynchronous event.
- `Module:handle_event(Event, StateName, StateData) -> Result`
  [page 169] Handle an asynchronous event.
- `Module:StateName(Event, From, StateData) -> Result`
  [page 169] Handle a synchronous event.
- `Module:handle_sync_event(Event, From, StateName, StateData) -> Result`
  [page 170] Handle a synchronous event.
- `Module:handle_info(Info, StateName, StateData) -> Result`
  [page 170] Handle an incoming message.
- `Module:terminate(Reason, StateName, StateData)`
  [page 171] Clean up before termination.
- `Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}`
  [page 171] Update the state data due to code replacement.

gen_server

The following functions are exported:

- `start(Module, Args, Options) -> Result`
  [page 173] Create a generic server process.
- `start(ServerName, Module, Args, Options) -> Result`
  [page 173] Create a generic server process.
- `start_link(Module, Args, Options) -> Result`
  [page 173] Create a generic server process.
- `start_link(ServerName, Module, Args, Options) -> Result`
  [page 173] Create a generic server process.
- `call(ServerRef, Request) -> Reply`
  [page 174] Make a synchronous call to a generic server.
- `call(ServerRef, Request, Timeout) -> Reply`
  [page 174] Make a synchronous call to a generic server.
- `multi_call(Name, Request) -> Result`
  [page 175] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request) -> Result`
  [page 175] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request, Timeout) -> Result`
  [page 175] Make a synchronous call to several generic servers.

- `cast(ServerRef, Request) -> ok`
  [page 176] Send an asynchronous request to a generic server.
- `abcast(Name, Request) -> abcast`
  [page 177] Send an asynchronous request to several generic servers.
- `abcast(Nodes, Name, Request) -> abcast`
  [page 177] Send an asynchronous request to several generic servers.
- `reply(Client, Reply) -> true`
  [page 177] Send a reply to a client.
- `enter_loop(Module, Options, State, ServerName, Timeout) ->`
  [page 177] Enters the gen_server receive loop.
- `enter_loop(Module, Options, State, Timeout) ->`
  [page 177] Enters the gen_server receive loop.
- `enter_loop(Module, Options, State, ServerName) -> _`
  [page 177] Enters the gen_server receive loop.
- `Module:init(Args) -> Result`
  [page 178] Initialize process and internal state.
- `Module:handle_call(Request, From, State) -> Result`
  [page 178] Handle a synchronous request.
- `Module:handle_cast(Request, State) -> Result`
  [page 179] Handle an asynchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 179] Handle an incoming message.
- `Module:terminate(Reason, State)`
  [page 180] Clean up before termination.
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 180] Update the internal state due to code replacement.

io

The following functions are exported:

- `put_chars([IoDevice,] Chars)`
  [page 182] Write characters to standard output
- `nl([IoDevice])`
  [page 182] Output a newline
- `get_chars([IoDevice,] Prompt, Count)`
  [page 182] Read characters from standard input
- `get_line([IoDevice,] Prompt)`
  [page 182] Read a line from standard input
- `setopts([IoDevice,] OptList)`
  [page 182] Set options for standard input/output
- `write([IoDevice,] Term)`
  [page 183] Write a term
- `read([IoDevice,] Prompt)`
  [page 183] Read a term
- `fwrite(Format)`
  [page 183] Write formatted output

io_lib

The following functions are exported:

- `write_atom(Atom)`
   Return an atom
- `write_string(String)`
   Return a string
- `write_char(Integer)`
   Return a character
- `indentation(String, StartIndent)`
   Indentation after printing string
- `char_list(CharList) -> bool()`
   Test for a list of characters
- `deep_char_list(CharList)`
   Test for a deep list of characters
- `printable_list(CharList)`
   Test for a list of printable characters

lib

The following functions are exported:

- `flush_receive() -> void()`
   Flush messages
- `error_message(Format, Args)`
   Print error message
- `progname() -> atom()`
   Return Erlang starter
- `nonl(List1)`
   Remove last newline
- `send(To, Msg)`
   Send a message
- `sendw(To, Msg)`
   Send a message and waits fo an answer

lists

The following functions are exported:

- `append(ListOfLists) -> List1`
   Append a list of lists
- `append(List1, List2) -> List3`
   Append two lists
- `concat(Things) -> string()`
   Concatenate a list of atoms
- `delete(Element, List1) -> List2`
   Delete an element in a list
- `duplicate(N, Element) -> List`
   Make N copies of element
- `flatlength(DeepList) -> int()`
   Length of flattened deep list

- `flatten(DeepList) -> List`
  [page 196] Flatten a deep list
- `flatten(DeepList, Tail) -> List`
  [page 196] Flatten a deep list
- `keydelete(Key, N, TupleList1) -> TupleList2`
  [page 197] Delete a tuple for a tuple list
- `keymember(Key, N, TupleList) -> bool()`
  [page 197] Test for a key in a list of tuples
- `keymerge(N, List1, List2)`
  [page 197] Merge two key-sorted lists
- `keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2`
  [page 197] Replace tuple in tuple list
- `keysearch(Key, N, TupleList) -> Result`
  [page 197] Extract value of key in a list of tuples
- `keysort(N, List1) -> List2`
  [page 197] Sort a list by key
- `last(List) -> Element`
  [page 198] Return last element in a list
- `max(List) -> Max`
  [page 198] Return maximum element of list
- `member(Element, List) -> bool()`
  [page 198] Test for membership of a list
- `merge(ListOfLists) -> List1`
  [page 198] Merge a list of sorted lists
- `merge(List1, List2) -> List3`
  [page 198] Merge two sorted lists
- `merge(Fun, List1, List2) -> List`
  [page 198] Merge two sorted list
- `merge3(List1, List2, List3) -> List4`
  [page 199] Merge three sorted lists
- `min(List) -> Min`
  [page 199] Return minimum element of list
- `nth(N, List) -> Element`
  [page 199] Extract element from a list
- `nthtail(N, List1) -> List2`
  [page 199] Return the N'th tail in List1
- `prefix(List1, List2) -> bool()`
  [page 199] Test for list prefix
- `reverse(List1) -> List2`
  [page 199] Reverse a list
- `reverse(List1, List2) -> List3`
  [page 200] Reverse a list appending a tail
- `seq(From, To) -> [int()]`
  [page 200] Generate a sequence of integers
- `seq(From, To, Incr) -> [int()]`
  [page 200] Generate a sequence of integers

- map(Func, List1) -> List2
  [page 204] Map a function over a list
- mapfoldl(Function, Acc0, List1) -> {List2, Acc}
  [page 204] Map and fold in one pass
- mapfoldr(Function, Acc0, List1) -> {List2, Acc}
  [page 205] Map and fold in one pass
- split(N, List) -> {List1, List2}
  [page 205] Partition List into one list of length N and one with the rest of the elements
- splitwith(Pred, List) -> {List1, List2}
  [page 205] Partition List1 into two lists according to Pred
- takewhile(Pred, List1) -> List2
  [page 205] Take elements from List1 while Pred is true

log_mf_h

The following functions are exported:

- init(Dir, MaxBytes, MaxFiles)
  [page 207] Initiate the event handler
- init(Dir, MaxBytes, MaxFiles, Pred) -> Args
  [page 207] Initiate the event handler

math

The following functions are exported:

- pi() -> float()
  [page 208] A useful number
- sin(X)
  [page 208] Diverse math functions
- cos(X)
  [page 208] Diverse math functions
- tan(X)
  [page 208] Diverse math functions
- asin(X)
  [page 208] Diverse math functions
- acos(X)
  [page 208] Diverse math functions
- atan(X)
  [page 208] Diverse math functions
- atan2(Y, X)
  [page 208] Diverse math functions
- sinh(X)
  [page 208] Diverse math functions
- cosh(X)
  [page 208] Diverse math functions

ms_transform

The following functions are exported:

orddict

No functions are exported.

ordsets

No functions are exported.

pg

The following functions are exported:

- `create(PgName)`
  [page 223] Create an empty group
- `create(PgName, Node)`
  [page 223] Create an epmty group on a node
- `join(PgName, Pid)`
  [page 223] Join a Pid to a process group
- `send(Pgname, Message)`
  [page 223] Send a message tuple to all members of a process group
- `esend(PgName, Mess)`
  [page 223] Send a message tuple to all members of a process group except the current node
- `members(PgName)`
  [page 223] >Return a list of the current members in the process group

pool

The following functions are exported:

- `start(Name)`
  [page 224] >Start a new pool
- `start(Name, Args)`
  [page 224] >Start a new pool
- `attach(Node)`
  [page 224] Ensure that a pool master is running
- `stop()`
  [page 224] Stop the pool and kill all the slave nodes
- `get_nodes()`
  [page 225] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args)`
  [page 225] Spawn a process on the expected lowest future loaded pool node
- `pspawn_link(Mod, Fun, Args)`
  [page 225] Spawn links a process on the expected lowest future loaded pool node
- `get_node()`
  [page 225] Return the node ID of the expected lowest future loaded node
- `new_node(Host, Name)`
  [page 225] Start a new node and attach it to an already existing pool

proc_lib

The following functions are exported:

- `spawn(Fun) -> Pid`
  [page 226] Spawn a new process.
- `spawn(Node,Fun) -> Pid`
  [page 226] Spawn a new process.
- `spawn(Module,Func,Args) -> Pid`
  [page 226] Spawn a new process.
- `spawn(Node,Module,Func,Args) -> Pid`
  [page 226] Spawn a new process.
- `spawn_link(Fun) -> Pid`
  [page 226] Spawn a new process and set a link.
- `spawn_link(Node,Fun) -> Pid`
  [page 226] Spawn a new process and set a link.
- `spawn_link(Module,Func,Args) -> Pid`
  [page 226] Spawn a new process and set a link.
- `spawn_link(Node,Module,Func,Args) -> Pid`
  [page 226] Spawn a new process and set a link.
- `spawn_opt(Fun,Opts) -> Pid`
  [page 227] Spawn a new process with given options.
- `spawn_opt(Node,Fun,Opts) -> Pid`
  [page 227] Spawn a new process with given options.
- `spawn_opt(Module,Func,Args,Opts) -> Pid`
  [page 227] Spawn a new process with given options.
- `spawn_opt(Node,Module,Func,Args,Opts) -> Pid`
  [page 227] Spawn a new process with given options.
- `start(Module,Func,Args) -> Ret`
  [page 227] Start a new process synchronously.
- `start(Module,Func,Args,Time) -> Ret`
  [page 227] Start a new process synchronously.
- `start(Module,Func,Args,Time,SpawnOpts) -> Ret`
  [page 227] Start a new process synchronously.
- `start_link(Module,Func,Args) -> Ret`
  [page 227] Start a new process synchronously.
- `start_link(Module,Func,Args,Time) -> Ret`
  [page 227] Start a new process synchronously.
- `start_link(Module,Func,Args,Time,SpawnOpts) -> Ret`
  [page 227] Start a new process synchronously.
- `init_ack(Parent, Ret) -> void()`
  [page 228] Used by a process when it has started.
- `init_ack(Ret) -> void()`
  [page 228] Used by a process when it has started.
- `format(CrashReport) -> string()`
  [page 228] Format a crash report.

- initial_call(PidOrPinfo) -> {Module,Function,Args} | Fun | false
  [page 229] Extract the initial call of a proc_lib spawned process.
- translate_initial_call(PidOrPinfo) -> {Module,Function,Arity} | Fun
  [page 229] Extract and translate the initial call of a proc_lib spawned process.
- hibernate(Module, Function, Arguments)
  [page 229] Hibernate the current process until a message is sent to it

proplists

The following functions are exported:

- append_values(Key, List) -> List
  [page 231]
- compact(List) -> List
  [page 231]
- delete(Key, List) -> List
  [page 231]
- expand(Expansions, List) -> List
  [page 231]
- get_all_values(Key, List) -> [term()]
  [page 232]
- get_bool(Key, List) -> bool()
  [page 232]
- get_keys(List) -> [term()]
  [page 232]
- get_value(Key, List) -> term()
  [page 233]
- get_value(Key, List, Default) -> term()
  [page 233]
- is_defined(Key, List) -> bool()
  [page 233]
- lookup(Key, List) -> none | tuple()
  [page 233]
- lookup_all(Key, List) -> [tuple()]
  [page 233]
- normalize(List, Stages) -> List
  [page 233]
- property(Property) -> Property
  [page 234]
- property(Key, Value) -> Property
  [page 234]
- split(List, Keys) -> {Lists, Rest}
  [page 234]
- substitute_aliases(Aliases, List) -> List
  [page 235]
- substitute_negations(Negations, List) -> List
  [page 235]
- unfold(List) -> List
  [page 235]

queue

The following functions are exported:

- `cons(Item, Q1) -> Q2`
  [page 236] Insert an item at the head of a queue
- `daeh(Q) -> Item`
  [page 236] Return the last item of a queue
- `from_list(L) -> queue()`
  [page 236] Convert a list to a queue
- `head(Q) -> Item`
  [page 236] Return the item at the head of a queue
- `in(Item, Q1) -> Q2`
  [page 236] Insert an item at the tail of a queue
- `in_r(Item, Q1) -> Q2`
  [page 237] Insert an item at the head of a queue
- `init(Q1) -> Q2`
  [page 237] Remove the last item from a queue
- `is_empty(Q) -> true | false`
  [page 237] Test if a queue is empty
- `join(Q1, Q2) -> Q3`
  [page 237] Join two queues
- `lait(Q1) -> Q2`
  [page 237] Remove the last item from a queue
- `last(Q) -> Item`
  [page 237] Return the last item of a queue
- `len(Q) -> N`
  [page 237] Get the length of a queue
- `new() -> Q`
  [page 238] Create a new empty FIFO queue
- `out(Q1) -> Result`
  [page 238] Remove the head item from a queue
- `out_r(Q1) -> Result`
  [page 238] Remove the last item from a queue
- `reverse(Q1) -> Q2`
  [page 238] Reverse a queue
- `snoc(Q1, Item) -> Q2`
  [page 238] Insert an item at the end of a queue
- `split(N, Q1) -> {Q2,Q3}`
  [page 238] Split a queue in two
- `tail(Q1) -> Q2`
  [page 238] Remove the head item from a queue
- `to_list(Q) -> list()`
  [page 239] Convert a queue to a list

random

The following functions are exported:

- `seed() -> ran()`
  [page 240] Seeds random number generation with default values
- `seed(A1, A2, A3) -> ran()`
  [page 240] Seeds random number generator
- `seed0() -> ran()`
  [page 240] Return default state for random number generation
- `uniform()-> float()`
  [page 240] Return a random float
- `uniform(N) -> int()`
  [page 240] Return a random integer
- `uniform_s(State0) -> {float(), State1}`
  [page 241] Return a random float
- `uniform_s(N, State0) -> {int(), State1}`
  [page 241] Return a random integer

regexp

The following functions are exported:

- `match(String, RegExp) -> MatchRes`
  [page 242] Match a regular expression
- `first_match(String, RegExp) -> MatchRes`
  [page 242] Match a regular expression
- `matches(String, RegExp) -> MatchRes`
  [page 242] Match a regular expression
- `sub(String, RegExp, New) -> SubRes`
  [page 243] Substitute the first occurrence of a regular expression
- `gsub(String, RegExp, New) -> SubRes`
  [page 243] Substitute all occurrences of a regular expression
- `split(String, RegExp) -> SplitRes`
  [page 243] Split a string into fields
- `sh_to_awk(ShRegExp) -> AwkRegExp`
  [page 244] Convert an `sh` regular expression into an `AWK` one
- `parse(RegExp) -> ParseRes`
  [page 244] Parse a regular expression
- `format_error(ErrorDescriptor) -> string()`
  [page 244] Format an error descriptor

sets

The following functions are exported:

- `new() -> Set`
  [page 247] Return an empty set
- `is_set(Set) -> bool()`
  [page 247] Test for an `Set`
- `size(Set) -> int()`
  [page 247] Return the number of elements in a set
- `to_list(Set) -> List`
  [page 247] Convert an `Set` into a list
- `from_list(List) -> Set`
  [page 247] Convert a list into an `Set`
- `is_element(Element, Set) -> bool()`
  [page 247] Test for membership of an `Set`
- `add_element(Element, Set1) -> Set2`
  [page 248] Add an element to an `Set`
- `del_element(Element, Set1) -> Set2`
  [page 248] Remove an element from an `Set`
- `union(Set1, Set2) -> Set3`
  [page 248] Return the union of two `Sets`
- `union(SetList) -> Set`
  [page 248] Return the union of a list of `Sets`
- `intersection(Set1, Set2) -> Set3`
  [page 248] Return the intersection of two `Sets`
- `intersection(SetList) -> Set`
  [page 248] Return the intersection of a list of `Sets`
- `subtract(Set1, Set2) -> Set3`
  [page 248] Return the difference of two `Sets`
- `is_subset(Set1, Set2) -> bool()`
  [page 249] Test for subset
- `fold(Function, Acc0, Set) -> Acc1`
  [page 249] Fold over set elements
- `filter(Pred, Set1) -> Set2`
  [page 249] Filter set elements

shell

The following functions are exported:

- `history(N) -> integer()`
  [page 257] Sets the number of previous commands to keep
- `results(N) -> integer()`
  [page 257] Sets the number of previous commands to keep
- `start_restricted(Module) -> ok`
  [page 257] Exits a normal shell and starts a restricted shell.
- `stop_restricted() -> ok`
  [page 257] Exits a restricted shell and starts a normal shell.

shell_default

No functions are exported.

slave

The following functions are exported:

- `start(Host)`
  [page 259] Start a slave node at Host
- `start_link(Host)`
  [page 259] Start a slave node at Host
- `start(Host, Name)`
  [page 259] Start a slave node at Host called Name@Host
- `start_link(Host, Name)`
  [page 260] Start a slave node at Host called Name@Host
- `start(Host, Name, Args) -> {ok, Node} | {error, ErrorInfo}`
  [page 260] Start a slave node at Host called `Name@Host` and passes `Args` to new node
- `start_link(Host, Name, Args)`
  [page 260] Start a slave node at Host called Name@Host
- `stop(Node)`
  [page 261] Stop (kill) a node
- `pseudo([Master | ServerList])`
  [page 261] Start a number of pseudo servers
- `pseudo(Master, ServerList)`
  [page 261] Start a number of pseudo servers
- `relay(Pid)`
  [page 261] Run a pseudo server

sofs

The following functions are exported:

- `a_function(Tuples [, Type]) -> Function`
  [page 266] Create a function.
- `canonical_relation(SetOfSets) -> BinRel`
  [page 266] Return the canonical map.
- `composite(Function1, Function2) -> Function3`
  [page 266] Return the composite of two functions.
- `constant_function(Set, AnySet) -> Function`
  [page 266] Create the function that maps each element of a set onto another set.
- `converse(BinRel1) -> BinRel2`
  [page 267] Return the converse of a binary relation.
- `difference(Set1, Set2) -> Set3`
  [page 267] Return the difference of two sets.
- `digraph_to_family(Graph [, Type]) -> Family`
  [page 267] Create a family from a directed graph.

- image(BinRel, Set1) -> Set2
  [page 274] Return the image of a set under a binary relation.
- intersection(SetOfSets) -> Set
  [page 274] Return the intersection of a set of sets.
- intersection(Set1, Set2) -> Set3
  [page 274] Return the intersection of two sets.
- intersection_of_family(Family) -> Set
  [page 274] Return the intersection of a family.
- inverse(Function1) -> Function2
  [page 274] Return the inverse of a function.
- inverse_image(BinRel, Set1) -> Set2
  [page 275] Return the inverse image of a set under a binary relation.
- is_a_function(BinRel) -> Bool
  [page 275] Test for a function.
- is_disjoint(Set1, Set2) -> Bool
  [page 275] Test for disjoint sets.
- is_empty_set(AnySet) -> Bool
  [page 275] Test for an empty set.
- is_equal(AnySet1, AnySet2) -> Bool
  [page 275] Test two sets for equality.
- is_set(AnySet) -> Bool
  [page 275] Test for an unordered set.
- is_sofs_set(Term) -> Bool
  [page 276] Test for an unordered set.
- is_subset(Set1, Set2) -> Bool
  [page 276] Test two sets for subset.
- is_type(Term) -> Bool
  [page 276] Test for a type.
- join(Relation1, I, Relation2, J) -> Relation3
  [page 276] Return the join of two relations.
- multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2
  [page 276] Return the multiple relative product of a tuple of binary relations and a relation.
- no_elements(ASet) -> NoElements
  [page 277] Return the number of elements of a set.
- partition(SetOfSets) -> Partition
  [page 277] Return the coarsest partition given a set of sets.
- partition(SetFun, Set) -> Partition
  [page 277] Return a partition of a set.
- partition(SetFun, Set1, Set2) -> {Set3, Set4}
  [page 277] Return a partition of a set.
- partition_family(SetFun, Set) -> Family
  [page 278] Return a family indexing a partition.
- product(TupleOfSets) -> Relation
  [page 278] Return the Cartesian product of a tuple of sets.
- product(Set1, Set2) -> BinRel
  [page 278] Return the Cartesian product of two sets.

- `projection(SetFun, Set1) -> Set2`
  [page 279] Return a set of substituted elements.
- `range(BinRel) -> Set`
  [page 279] Return the range of a binary relation.
- `relation(Tuples [, Type]) -> Relation`
  [page 279] Create a relation.
- `relation_to_family(BinRel) -> Family`
  [page 279] Create a family from a binary relation.
- `relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2`
  [page 280] Return the relative product of a tuple of binary relations and a binary relation.
- `relative_product(BinRel1, BinRel2) -> BinRel3`
  [page 280] Return the relative product of two binary relations.
- `relative_product1(BinRel1, BinRel2) -> BinRel3`
  [page 280] Return the relative_product of two binary relations.
- `restriction(BinRel1, Set) -> BinRel2`
  [page 280] Return a restriction of a binary relation.
- `restriction(SetFun, Set1, Set2) -> Set3`
  [page 281] Return a restriction of a set.
- `set(Terms [, Type]) -> Set`
  [page 281] Create a set of atoms or any type of sets.
- `specification(Fun, Set1) -> Set2`
  [page 281] Select a subset using a predicate.
- `strict_relation(BinRel1) -> BinRel2`
  [page 281] Return the strict relation corresponding to a given relation.
- `substitution(SetFun, Set1) -> Set2`
  [page 282] Return a function with a given set as domain.
- `symdiff(Set1, Set2) -> Set3`
  [page 282] Return the symmetric difference of two sets.
- `symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`
  [page 283] Return a partition of two sets.
- `to_external(AnySet) -> ExternalSet`
  [page 283] Return the elements of a set.
- `to_sets(ASet) -> Sets`
  [page 283] Return a list or a tuple of the elements of set.
- `type(AnySet) -> Type`
  [page 283] Return the type of a set.
- `union(SetOfSets) -> Set`
  [page 283] Return the union of a set of sets.
- `union(Set1, Set2) -> Set3`
  [page 283] Return the union of two sets.
- `union_of_family(Family) -> Set`
  [page 284] Return the union of a family.
- `weak_relation(BinRel1) -> BinRel2`
  [page 284] Return the weak relation corresponding to a given relation.

string

The following functions are exported:

- `len(String) -> Length`
  [page 285] Return the length of a string
- `equal(String1, String2) -> bool()`
  [page 285] Test string equality
- `concat(String1, String2) -> String3`
  [page 285] Concatenate two strings
- `chr(String, Character) -> Index`
  [page 285] Return the index of the first/last occurrence of `Character` in `String`
- `rchr(String, Character) -> Index`
  [page 285] Return the index of the first/last occurrence of `Character` in `String`
- `str(String, SubString) -> Index`
  [page 285] Find the index of a substring
- `rstr(String, SubString) -> Index`
  [page 285] Find the index of a substring
- `span(String, Chars) -> Length`
  [page 286] Span characters at start of string
- `cspan(String, Chars) -> Length`
  [page 286] Span characters at start of string
- `substr(String, Start) -> SubString`
  [page 286] Return a substring of `String`
- `substr(String, Start, Length) -> Substring`
  [page 286] Return a substring of `String`
- `tokens(String, SeparatorList) -> Tokens`
  [page 286] Split string into tokens
- `chars(Character, Number) -> String`
  [page 286] Returns a string consisting of numbers of characters
- `chars(Character, Number, Tail) -> String`
  [page 286] Returns a string consisting of numbers of characters
- `copies(String, Number) -> Copies`
  [page 287] Copy a string
- `words(String) -> Count`
  [page 287] Count blank separated words
- `words(String, Character) -> Count`
  [page 287] Count blank separated words
- `sub_word(String, Number) -> Word`
  [page 287] Extract subword
- `sub_word(String, Number, Character) -> Word`
  [page 287] Extract subword
- `strip(String) -> Stripped`
  [page 287] Strip leading or trailing characters
- `strip(String, Direction) -> Stripped`
  [page 287] Strip leading or trailing characters

- `strip(String, Direction, Character) -> Stripped`
  [page 287] Strip leading or trailing characters
- `left(String, Number) -> Left`
  [page 288] Adjust left end of string
- `left(String, Number, Character) -> Left`
  [page 288] Adjust left end of string
- `right(String, Number) -> Right`
  [page 288] Adjust right end of string
- `right(String, Number, Character) -> Right`
  [page 288] Adjust right end of string
- `centre(String, Number) -> Centered`
  [page 288] Center a string
- `centre(String, Number, Character) -> Centered`
  [page 288] Center a string
- `sub_string(String, Start) -> SubString`
  [page 288] Extract a substring
- `sub_string(String, Start, Stop) -> SubString`
  [page 289] Extract a substring

supervisor

The following functions are exported:

- `start_link(Module, Args) -> Result`
  [page 292] Create a supervisor process.
- `start_link(SupName, Module, Args) -> Result`
  [page 292] Create a supervisor process.
- `start_child(SupRef, ChildSpec) -> Result`
  [page 292] Dynamically add a child process to a supervisor.
- `terminate_child(SupRef, Id) -> Result`
  [page 293] Terminate a child process belonging to a supervisor.
- `delete_child(SupRef, Id) -> Result`
  [page 294] Delete a child specification from a supervisor.
- `restart_child(SupRef, Id) -> Result`
  [page 294] Restart a terminated child process belonging to a supervisor.
- `which_children(SupRef) -> [{Id,Child,Type,Modules}]`
  [page 295] Return information about all children specifications and child processes belonging to a supervisor.
- `check_childspecs([ChildSpec]) -> Result`
  [page 295] Check if child specifications are syntactically correct.
- `Module:init(Args) -> Result`
  [page 296] Return a supervisor specification.

supervisor_bridge

The following functions are exported:

- `start_link(Module, Args) -> Result`
  [page 297] Create a supervisor bridge process.
- `start_link(SupBridgeName, Module, Args) -> Result`
  [page 297] Create a supervisor bridge process.
- `Module:init(Args) -> Result`
  [page 298] Initialize process and start subsystem.
- `Module:terminate(Reason, State)`
  [page 298] Clean up and stop subsystem.

sys

The following functions are exported:

- `log(Name,Flag)`
  [page 301] Log system events in memory
- `log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}`
  [page 301] Log system events in memory
- `log_to_file(Name,Flag)`
  [page 301] Log system events to the specified file
- `log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`
  [page 301] Log system events to the specified file
- `statistics(Name,Flag)`
  [page 301] Enable or disable the collections of statistics
- `statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`
  [page 301] Enable or disable the collections of statistics
- `trace(Name,Flag)`
  [page 302] Print all system events on `standard_io`
- `trace(Name,Flag,Timeout) -> void()`
  [page 302] Print all system events on `standard_io`
- `no_debug(Name)`
  [page 302] Turn off debugging
- `no_debug(Name,Timeout) -> void()`
  [page 302] Turn off debugging
- `suspend(Name)`
  [page 302] Suspend the process
- `suspend(Name,Timeout) -> void()`
  [page 302] Suspend the process
- `resume(Name)`
  [page 302] Resume a suspended process
- `resume(Name,Timeout) -> void()`
  [page 302] Resume a suspended process
- `change_code(Name, Module, OldVsn, Extra)`
  [page 302] Send the code change system message to the process

- change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}
  [page 302] Send the code change system message to the process
- get_status(Name)
  [page 302] Get the status of the process
- get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}
  [page 302] Get the status of the process
- install(Name,{Func,FuncState})
  [page 303] Install a debug function in the process
- install(Name,{Func,FuncState},Timeout)
  [page 303] Install a debug function in the process
- remove(Name,Func)
  [page 303] Remove a debug function from the process
- remove(Name,Func,Timeout) -> void()
  [page 303] Remove a debug function from the process
- debug_options(Options) -> [dbg_opt()]
  [page 304] Convert a list of options to a debug structure
- get_debug(Item,Debug,Default) -> term()
  [page 304] Get the data associated with a debug option
- handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]
  [page 304] Generate a system event
- handle_system_msg(Msg,From,Parent,Module,Debug,Misc)
  [page 304] Take care of system messages
- print_log(Debug) -> void()
  [page 305] Print the logged events in the debug structure
- Mod:system_continue(Parent, Debug, Misc)
  [page 305] Called when the process should continue its execution
- Mod:system_terminate(Reason, Parent, Debug, Misc)
  [page 305] Called when the process should terminate
- Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}
  [page 305] Called when the process should perform a code change

timer

The following functions are exported:

- start() -> ok
  [page 307] Start a global timer server (named timer_server).
- apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}
  [page 307] Apply Module:Function(Arguments) after a specified Time.
- send_after(Time, Pid, Message) -> {ok, TRef} | {error,Reason}
  [page 307] Send Message to Pid after a specified Time.
- send_after(Time, Message) -> {ok, TRef} | {error,Reason}
  [page 307] Send Message to Pid after a specified Time.

- `exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}`
  [page 308] Send an exit signal with Reason after a specified Time.
- `exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}`
  [page 308] Send an exit signal with Reason after a specified Time.
- `kill_after(Time, Pid)-> {ok, TRef} | {error,Reason2}`
  [page 308] Send an exit signal with Reason after a specified Time.
- `kill_after(Time) -> {ok, TRef} | {error,Reason2}`
  [page 308] Send an exit signal with Reason after a specified Time.
- `apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`
  [page 308] Evaluate Module:Function(Arguments) repeatedly at intervals of Time.
- `send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 308] Send Message repeatedly at intervals of Time.
- `send_interval(Time, Message) -> {ok, TRef} | {error, Reason}`
  [page 308] Send Message repeatedly at intervals of Time.
- `cancel(TRef) -> {ok, cancel} | {error, Reason}`
  [page 308] Cancel a previously requested timeout identified by TRef.
- `sleep(Time) -> ok`
  [page 308] Suspend the calling process for Time amount of milliseconds.
- `tc(Module, Function, Arguments) -> {Time, Value}`
  [page 309] Measure the real time it takes to evaluate apply(Module, Function, Arguments)
- `now_diff(T2, T1) -> {Time, Value}`
  [page 309] Calculate time difference between now/0 timestamps
- `seconds(Seconds) -> Milliseconds`
  [page 309] Convert Seconds to Milliseconds.
- `minutes(Minutes) -> Milliseconds`
  [page 309] Converts Minutes to Milliseconds.
- `hours(Hours) -> Milliseconds`
  [page 309] Convert Hours to Milliseconds.
- `hms(Hours, Minutes, Seconds) -> Milliseconds`
  [page 309] Convert Hours+Minutes+Seconds to Milliseconds.

win32reg

The following functions are exported:

- `change_key(RegHandle, Key) -> ReturnValue`
  [page 312] Move to a key in the registry
- `change_key_create(RegHandle, Key) -> ReturnValue`
  [page 312] Move to a key, create it if it is not there
- `close(RegHandle)-> ReturnValue`
  [page 312] Close the registry.
- `current_key(RegHandle) -> ReturnValue`
  [page 312] Return the path to the current key.

- `delete_key(RegHandle) -> ReturnValue`
  [page 312] Delete the current key
- `delete_value(RegHandle, Name) -> ReturnValue`
  [page 313] Delete the named value on the current key.
- `expand(String) -> ExpandedString`
  [page 313] Expand a string with environment variables
- `format_error(ErrorId) -> ErrorString`
  [page 313] Convert an POSIX errorcode to a string
- `open(OpenModeList)-> ReturnValue`
  [page 313] Open the registry for reading or writing
- `set_value(RegHandle, Name, Value) -> ReturnValue`
  [page 313] Set value at the current registry key with specified name.
- `sub_keys(RegHandle) -> ReturnValue`
  [page 314] Get subkeys to the current key.
- `value(RegHandle, Name) -> ReturnValue`
  [page 314] Get the named value on the current key.
- `values(RegHandle) -> ReturnValue`
  [page 314] Get all values on the current key.

beam_lib

Erlang Module

beam_lib provides an interface to files created by the BEAM compiler ("BEAM files"). The format used, a variant of "EA IFF 1985" Standard for Interchange Format Files, divides data into chunks.

Chunk data can be returned as binaries or as compound terms. Compound terms are returned when chunks are referenced by names (atoms) rather than identifiers (strings). The names recognized and the corresponding identifiers are abstract_code ("Abst"), attributes ("Attr"), exports ("ExpT"), labeled_exports ("ExpT"), imports ("ImpT"), locals ("LocT"), labeled_locals ("LocT"), and atoms ("Atom").

The syntax of the compound term (ChunkData) is as follows:

- ChunkData = {ChunkId, binary()} | {abstract_code, AbstractCode} | {attributes, [{Attribute, [AttributeValue]}]} | {exports, [{Function, Arity}]} | {labeled_exports, [{Function, Arity, Label}]} | {imports, [{Module, Function, Arity}]} | {locals, [{Function, Arity}]}]} | {labeled_locals, [{Function, Arity, Label}]}]} | {atoms, [{integer(), atom()}]}
- ChunkRef = ChunkId | ChunkName
- ChunkName = abstract_code | attributes | exports | labeled_exports | imports | locals | labeled_locals | atoms
- ChunkId = string()
- AbstractCode = {AbstVersion, forms()} | no_abstract_code
- AbstVersion = atom()
- Attribute = atom()
- AttributeValue = term()
- Module = Function = atom()
- Arity = integer() >= 0
- Label = integer() >= 0

The list of attributes is sorted on Attribute, and each attribute name occurs once in the list. The attribute values occur in the same order as on the file. The lists of functions are also sorted. It is not checked that the forms conform to the abstract format indicated by AbstVersion.

no_abstract_code means that the "Abst" chunk is present, but empty.

Each of the functions described below accept either a filename or a binary containing a beam module.

Exports

`chunks(FileNameOrBinary, [ChunkRef]) -> {ok, {Module, [ChunkData]}} | {error, Module, Reason}`

Types:

- FileNameOrBinary = string() | atom() | binary()
- Reason = {unknown_chunk, FileName, atom()} | -see info/1-

The `chunks/2` function reads chunk data for selected chunks. The order of the returned list of chunk data is determined by the order of the list of chunks references; if each chunk data were replaced by the tag, the result would be the given list.

`version(FileNameOrBinary) -> {ok, {Module, Version}} | {error, Module, Reason}`

Types:

- FileNameOrBinary = string() | atom() | binary()
- Version = [term()]
- Reason = -see chunks/2-

The `version/1` function returns the module version(s) found in a BEAM file.

`info(FileNameOrBinary) -> [SourceRef, {module, Module}, {chunks, [ChunkInfo]}] | {error, Module, Reason}`

Types:

- FileName = string() | atom()
- FileNameOrBinary = FileName | binary()
- SourceRef = {file, FileName} | {binary, binary()}
- ChunkInfo = {ChunkId, StartPosition, Size}
- StartPosition = integer() > 0
- Size = integer() >= 0
- Reason = {chunk_too_big, FileName, ChunkId, ChunkSize, FileSize} | {invalid_beam_file, FileName, FilePosition} | {invalid_chunk, FileName, ChunkId} | {missing_chunk, FileName, ChunkId} | {not_a_beam_file, FileName} | {file_error, FileName, FileError}

The `info/1` function extracts some information about a BEAM file: the file name, the module name, and for each chunk the identifier as well as the position and size in bytes of the chunk data.

`cmp(FileNameOrBinary, FileNameOrBinary) -> ok | {error, Module, Reason}`

Types:

- FileName = string() | atom()
- FileNameOrBinary = FileName | binary()
- Reason = {modules_different, Module, Module} | {chunks_different, ChunkId} | -see info/1-

The `cmp/2` function compares the contents of two BEAM files. If the module names are the same, and the chunks with the identifiers "Code", "ExpT", "ImpT", "StrT", and "Atom" have the same contents in both files, `ok` is returned. Otherwise an error message is returned.

`cmp_dirs(Directory1, Directory2) -> {Only1, Only2, Different} | {error, Module, Reason}`

Types:

- Directory1 = Directory2 = string() | atom()
- Different = [{FileName1, FileName2}]
- Only1 = Only2 = [FileName]
- FileName = FileName1 = FileName2 = string()
- Reason = -see info/1-

The `cmp_dirs/2` function compares the BEAM files in two directories. Only files with extension ".beam" are compared. BEAM files that exist in directory `Directory1` (`Directory2`) only are returned in `Only1` (`Only2`). BEAM files that exist on both directories but are considered different by `cmp/2` are returned as pairs {`FileName1`, `FileName2`} where `FileName1` (`FileName2`) exists in directory `Directory1` (`Directory2`).

`diff_dirs(Directory1, Directory2) -> ok | {error, Module, Reason}`

Types:

- Directory1 = Directory2 = string() | atom()
- Reason = -see info/1-

The `diff_dirs/2` function compares the BEAM files in two directories the way `cmp_dirs/2` does, but names of files that exist in only one directory or are different are presented on standard output.

`strip(FileNameOrBinary) -> {ok, {Module, FileNameOrBinary}} | {error, Module, Reason}`

Types:

- FileName = string() | atom()
- FileNameOrBinary = FileName | binary()
- Reason = -see info/1-

The `strip/1` function removes all chunks from a BEAM file except those needed by the loader. In particular, the abstract code is removed. The module name found in the file and the file name, possibly with the ".beam" extension added, are returned.

`strip_files(Files) -> {ok, [{Module, FileNameOrBinary]}} | {error, Module, Reason}`

Types:

- Files = [FileNameOrBinary]
- FileName = string() | atom()
- FileNameOrBinary = FileName | binary()
- Reason = -see info/1-

The `strip_files/1` function removes all chunks except those needed by the loader from BEAM files. In particular, the abstract code is removed. The returned list contains one element for each given file name, ordered as the given list. The list element is a pair of the module name found in the file and the file name, the latter possibly with the ".beam" extension added.

`strip_release(Directory) -> {ok, [{Module, FileName]}} | {error, Module, Reason}`

Types:

- Directory = string() | atom()
- FileName = string()
- Reason = -see info/1-

The `strip_release/1` function removes all chunks except those needed by the loader from the BEAM files of a release. `Directory` should be the installation root directory. For example, the current OTP release can be stripped with the call `beam_lib:strip_release(code:root_dir())`. The returned list contains module names and file names of stripped files.

`format_error(Error) -> character_list()`

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

c

Erlang 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).

`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`.

`memory() -> MemoryInformation`

The same as `erlang:memory/0`, see the `erlang(3)` man page.

`memory(MemoryTypeSpecification) -> MemoryInformation`

The same as `erlang:memory/1`, see the `erlang(3)` man page.

`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`.

`ni() -> void()`

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

`nl(Module) -> void()`

Types:

- Module = atom()

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

`nregs() -> void()`

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

`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.

`xm(ModSpec) -> void()`

Types:

- ModSpec = Module | File
- Module = atom()
- File = string()

This function finds undefined functions and unused functions in a module by calling `xref:m/1`.

`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.

See Also

erlang(3)

calendar

Erlang 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.

> **Warning:**
> This function is deprecated. Use `local_time_to_universal_time_dst/1` instead, as it gives a more correct and complete result. Especially for the period that does not exist since it gets skipped during the switch *to* daylight saving time, this function still returns a result.

`local_time_to_universal_time_dst(DateTime) -> [DateTimeUTC]`

Types:

- DateTime = DateTimeUTC{Date, Time}
- 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.

The return value is a list of 0, 1 or 2 possible UTC times:

`[]` For a local `DateTime` during the period that is skipped when switching *to* daylight saving time, there is no corresponding UTC since the local time is illegal - it has never happened.

`[DstDateTimeUTC,DateTimeUTC]` For a local `DateTime` during the period that is repeated when switching *from* daylight saving time, there are two corresponding UTCs. One for the first instance of the period when daylight saving time is still active, and one for the second instance.

`[DateTimeUTC]` For all other local times there is only one corresponding UTC.

`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_secnds(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

Erlang Module

The module `dets` provides a term storage on file. The stored terms, in this module called *objects*, are tuples such that one element is defined to be the key. A Dets *table* is a collection of objects with the key at the same position stored on a file.

Dets is used by the Mnesia application, and is provided as is for users who are interested in an efficient storage of Erlang terms on disk only. Many applications just need to store some terms in a file. Mnesia adds transactions, queries, and distribution. The size of Dets files cannot exceed 2 GB. If larger tables are needed, Mnesia's table fragmentation can be used.

There are three types of Dets tables: set, bag and duplicate_bag. A table of type *set* has at most one object with a given key. If an object with a key already present in the table is inserted, the existing object is overwritten by the new object. A table of type *bag* has zero or more different objects with a given key. A table of type *duplicate_bag* has zero or more possibly equal objects with a given key.

Dets tables must be opened before they can be updated or read, and when finished they must be properly closed. If a table has not been properly closed, Dets will automatically repair the table. This can take a substantial time if the table is large. A Dets table is closed when the process which opened the table terminates. If several Erlang processes (users) open the same Dets table, they will share the table. The table is properly closed when all users have either terminated or closed the table. Dets tables are not properly closed if the Erlang runtime 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 performed by Dets are disk operations, it is important to realize that a single look-up operation involves a series of disk seek and read operations. For this reason, the Dets functions are much slower than the corresponding Ets functions, although Dets exports a similar interface.

Dets organizes data as a linear hash list and the hash list grows gracefully as more data is inserted into the table. Space management on the file is performed by what is called a buddy system. The current implementation keeps the entire buddy system in RAM, which implies that if the table gets heavily fragmented, quite some memory can be used up. The only way to defragment a table is to close it and then open it again with the `repair` option set to `force`.

It is worth noting that the ordered_set type present in Ets is not yet implemented by Dets, neither is the limited support for concurrent updates which makes a sequence of `first` and `next` calls safe to use on fixed Ets tables. Both these features will be implemented by Dets in a future release of Erlang/OTP. Until then, the Mnesia application (or some user implemented method for locking) has to be used to implement safe concurrency. Currently, no library of Erlang/OTP has support for ordered disk based term storage.

Two versions of the format used for storing objects on file are supported by Dets. The first version, 8, is the format always used for tables created by OTP R7 and earlier. The second version, 9, is the default version of tables created by OTP R8 (and later OTP releases). OTP R8 can create version 8 tables, and convert version 8 tables to version 9, and vice versa, upon request.

All Dets functions return `{error, Reason}` if an error occurs (`first/1` and `next/2` are exceptions, they exit the process with the error tuple). If given badly formed arguments, all functions exit the process with a `badarg` message.

Types

```
access() = read | read_write
auto_save() = infinity | int()
bindings_cont() = tuple()
bool() = true | false
file() = string()
int() = integer() >= 0
keypos() = integer() >= 1
name() = atom() | ref()
no_slots() = integer() >= 0 | default
object() = tuple()
object_cont() = tuple()
select_cont() = tuple()
type() = bag | duplicate_bag | set
version() = 8 | 9 | default
```

Exports

`all() -> [Name]`

Types:

- Name = name()

Returns a list of the names of all open tables on this node.

`bchunk(Name, Continuation) -> {Continuation2, Data} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Continuation = start | cont()
- Continuation2 = cont()
- Data = binary() | tuple()

Returns a list of objects stored in a table. The exact representation of the returned objects is not public. The lists of data can be used for initializing a table by giving the value `bchunk` to the `format` option of the `init_table/3` function. The Mnesia application uses this function for copying open tables.

Unless the table is protected using `safe_fixtable/2`, calls to `bchunk/2` may not work as expected if concurrent updates are made to the table.

The first time `bchunk/2` is called, an initial continuation, the atom `start`, must be provided.

The `bchunk/2` function returns a tuple `{Continuation2, Data}`, where `Data` is a list of objects. `Continuation2` is another continuation which is to be passed on to a subsequent call to `bchunk/2`. With a series of calls to `bchunk/2` it is possible to extract all objects of the table.

`bchunk/2` returns `'$end_of_table'` when all objects have been returned, or `{error, Reason}` if an error occurs.

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

Types:

- Name = name()

Closes a table. Only processes that have opened a table are allowed to close it.

All open tables must be closed before the system is stopped. If an attempt is made to open a table which has not been properly closed, Dets automatically tries to repair the table.

`delete(Name, Key) -> ok | {error, Reason}`

Types:

- Name = name()

Deletes all objects with the key `Key` from the table `Name`.

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

Types:

- Name = name()

Deletes all objects from a table in almost constant time. However, if the table if fixed, `delete_all_objects(T)` is equivalent to `match_delete(T, '_')`.

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

Types:

- Name = name()
- Object = object()

Deletes all instances of a given object from a table. If a table is of type `bag` or `duplicate_bag`, the `delete/2` function cannot be used to delete only some of the objects with a given key. This function makes this possible.

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

Types:

- Key = term()
- Name = name()

Returns the first key stored in the table `Name` according to the table's internal order, or `'$end_of_table'` if the table is empty.

Unless the table is protected using `safe_fixtable/2`, subsequent calls to `next/2` may not work as expected if concurrent updates are made to the table.

Should an error occur, the process is exited with an error tuple `{error, Reason}`. The reason for not returning the error tuple is that it cannot be distinguished from a key.

There are two reasons why `first/1` and `next/2` should not be used: they are not very efficient, and they prevent the use of the key `'$end_of_table'` since this atom is used to indicate the end of the table. If possible, the `match`, `match_object`, and `select` functions should be used for traversing tables.

`foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}`

Types:

- Function = fun(Object, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Name = name()
- Object = object()

Calls `Function` on successive elements of the table `Name` together with an extra argument `AccIn`. The order in which the elements of the table are traversed is unspecified. `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if the table is empty.

`foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}`

Types:

- Function = fun(Object, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Name = name()
- Object = object()

Calls `Function` on successive elements of the table `Name` together with an extra argument `AccIn`. The order in which the elements of the table are traversed is unspecified. `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if the table is empty.

`from_ets(Name, EtsTab) -> ok | {error, Reason}`

Types:

- Name = name()
- EtsTab = -see ets(3)-

Replaces the objects of the table `Name` with the objects of the Ets table `EtsTab`. The order in which the objects are inserted is not specified. Since `ets:safe_fixtable/2` is called, the Ets table must be public or owned by the calling process.

`info(Name) -> InfoList | undefined`

Types:

- Name = name()
- InfoList = [{Item, Value}]

Returns information about the table `Name` as a list of `{Item, Value}` tuples:

- `{file_size, int()}`, the size of the file in bytes.
- `{filename, file()}`, the name of the file where objects are stored.
- `{keypos, keypos()}`, the position of the key.
- `{size, int()}`, the number of objects stored in the table.
- `{type, type()}`, the type of the table.

`info(Name, Item) -> Value | undefined`

Types:

- Name = name()

Returns the information associated with `Item` for the table `Name`. In addition to the `{Item, Value}` pairs defined for `info/1`, the following items are allowed:

- `{access, access()}`, the access mode.
- `{auto_save, auto_save()}`, the auto save interval.
- `{bchunk_format, binary()}`, an opaque binary describing the format of the objects returned by `bchunk/2`. The binary can be used as argument to `is_compatible_chunk_format/2`. Only available for version 9 tables.
- `{hash`, Hash}. Describes which BIF is used to calculate the hash values of the objects stored in the Dets table. Possible values of Hash are `hash`, which implies that the `erlang:hash/2` BIF is used, `phash`, which implies that the `erlang:phash/2` BIF is used, and `phash2`, which implies that the `erlang:phash2/1` BIF is used.
- `{memory, int()}`, the size of the file in bytes. The same value is associated with the item `file_size`.
- `{no_keys, int()}`, the number of different keys stored in the table. Only available for version 9 tables.
- `{no_objects, int()}`, the number of objects stored in the table.
- `{no_slots, {Min, Used, Max}}`, the number of slots of the table. `Min` is the minimum number of slots, `Used` is the number of currently used slots, and `Max` is the maximum number of slots. Only available for version 9 tables.
- `{owner, pid()}`, the pid of the process that handles requests to the Dets table.
- `{ram_file, bool()}`, whether the table is kept in RAM.
- `{safe_fixed`, SafeFixed}. If the table is fixed, SafeFixed is a tuple `{FixedAtTime, [{Pid,RefCount}]}`. `FixedAtTime` is the time when the table was first fixed, and `Pid` is the pid of the process that fixes the table `RefCount` times. There may be any number of processes in the list. If the table is not fixed, SafeFixed is the atom `false`.
- `{version, int()}`, the version of the format of the table.

`init_table(Name, InitFun [, Options]) -> ok | {error, Reason}`

Types:

- Name = atom()
- InitFun = fun(Arg) -> Res
- Arg = read | close
- Res = end_of_input | {[object()], InitFun} | {Data, InitFun} | term()
- Data = binary() | tuple()

Replaces the existing objects of the table `Name` with objects created by calling the input function `InitFun`, see below. The reason for using this function rather than calling `insert/2` is that of efficiency. It should be noted that the input functions are called by the process that handles requests to the Dets table, not by the calling process.

When called with the argument `read` the function `InitFun` is assumed to return `end_of_input` when there is no more input, or `{Objects, Fun}`, where `Objects` is a list of objects and `Fun` is a new input function. Any other value Value is returned as an error `{error, {init_fun, Value}}`. Each input function will be called exactly once, and should an error occur, the last function is called with the argument `close`, the reply of which is ignored.

If the type of the table is `set` and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. Extra objects should be avoided, or the file will be unnecessarily fragmented. This holds also for duplicated objects stored in tables of type `duplicate_bag`.

It is important that the table has a sufficient number of slots for the objects. If not, the hash list will start to grow when `init_table/2` returns which will significantly slow down access to the table for a period of time. The minimum number of slots is set by the `open_file/2` option `min_no_slots` and returned by the `info/2` item `no_slots`. See also the `min_no_slots` option below.

The `Options` argument is a list of `{Key, Val}` tuples where the following values are allowed:

- `{min_no_slots, no_slots()}`. Specifies the estimated number of different keys that will be stored in the table. The `open_file` option with the same name is ignored unless the table is created, and in that case performance can be enhanced by supplying an estimate when initializing the table.
- `{format, Format}`. Specifies the format of the objects returned by the function `InitFun`. If `Format` is `term` (the default), `InitFun` is assumed to return a list of tuples. If `Format` is `bchunk`, `InitFun` is assumed to return `Data` as returned by `bchunk/2`. This option overrides the `min_no_slots` option.

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

Types:

- Name = name()
- Objects = object() | [object()]

Inserts one or more objects into the table `Name`. If there already exists an object with the same key as some of the given objects and the table type is `set`, the old object will be replaced.

`insert_new(Name, Objects) -> Bool`

Types:

- Name = name()
- Objects = object() | [object()]
- Bool = bool()

Inserts one or more objects into the table `Name`. If there already exists an object with the same key as some of the given objects the table is not updated and `false` is returned, otherwise the objects are inserted and `true` returned.

`is_compatible_bchunk_format(Name, BchunkFormat) -> Bool`

Types:

- Name = name()
- BchunkFormat = binary()
- Bool = bool()

Returns `true` if it would be possible to initialize the table `Name`, using `init_table/3` with the option `{format,bchunk}`, with objects read with `bchunk/2` from some table `T` such that calling `info(T,bchunk_format)` returns `BchunkFormat`.

`is_dets_file(FileName) -> Bool | {error, Reason}`

Types:

- FileName = file()
- Bool = bool()

Returns `true` if the file `FileName` is a Dets table, `false` otherwise.

`lookup(Name, Key) -> [Object] | {error, Reason}`

Types:

- Key = term()
- Name = name()
- Object = object()

Returns a list of all objects with the key `Key` stored in the table `Name`. 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 the empty list or a list with one object, as there cannot be more than one object with a given key. If the table is of type `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Note that the order of objects returned is unspecified. In particular, the order in which objects were inserted is not reflected.

`match(Continuation) -> {[Match], Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = bindings_cont()

- Match = [term()]

Matches some objects stored in a table and returns a list of the bindings that match a given pattern in some unspecified order. The table, the pattern, and the number of objects that are matched are all defined by `Continuation`, which has been returned by a prior call to `match/1` or `match/3`.

When all objects of the table have been matched, `'$end_of_table'` is returned.

`match(Name, Pattern) -> [Match] | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- Match = [term()]

Returns for each object of the table `Name` that matches `Pattern` a list of bindings in some unspecified order. See ets(3) [page 117] for a description of patterns. If the keypos'th element of `Pattern` is unbound, all objects of the table are matched. If the keypos'th element is bound, only the objects with the right key are matched.

`match(Name, Pattern, N) -> {[Match], Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- N = default | int()
- Match = [term()]
- Continuation = bindings_cont()

Matches some or all objects of the table `Name` and returns a list of the bindings that match `Pattern` in some unspecified order. See ets(3) [page 117] for a description of patterns.

A tuple of the bindings and a continuation is returned, unless the table is empty, in which case `'$end_of_table'` is returned. The continuation is to be used when matching further objects by calling `match/1`.

If the keypos'th element of `Pattern` is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, `N` objects at a time. The default, indicated by giving `N` the value `default`, is to let the number of objects vary depending on the sizes of the objects. If `Name` is a version 9 table, all objects with the same key are always matched at the same time which implies that more than N objects may sometimes be matched.

The table should always be protected using `safe_fixtable/2` before calling `match/3`, or errors may occur when calling `match/1`.

`match_delete(Name, Pattern) -> N | {error, Reason}`

Types:

- Name = name()
- N = int()
- Pattern = tuple()

Deletes all objects that match `Pattern` from the table `Name`, and returns the number of deleted objects. See ets(3) [page 117] for a description of patterns.

If the keypos'th element of `Pattern` is bound, only the objects with the right key are matched.

`match_object(Continuation) -> {[Object], Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = object_cont()
- Object = object()

Returns a list of some objects stored in a table that match a given pattern in some unspecified order. The table, the pattern, and the number of objects that are matched are all defined by `Continuation`, which has been returned by a prior call to `match_object/1` or `match_object/3`.

When all objects of the table have been matched, `'$end_of_table'` is returned.

`match_object(Name, Pattern) -> [Object] | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- Object = object()

Returns a list of all objects of the table `Name` that match `Pattern` in some unspecified order. See ets(3) [page 117] for a description of patterns.

If the keypos'th element of `Pattern` is unbound, all objects of the table are matched. If the keypos'th element of `Pattern` is bound, only the objects with the right key are matched.

Using the `match_object` functions for traversing all objects of a table is more efficient than calling `first/1` and `next/2` or `slot/2`.

`match_object(Name, Pattern, N) -> {[Object], Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- N = default | int()
- Object = object()
- Continuation = object_cont()

Matches some or all objects stored in the table `Name` and returns a list of the objects that match `Pattern` in some unspecified order. See ets(3) [page 117] for a description of patterns.

A list of objects and a continuation is returned, unless the table is empty, in which case `'$end_of_table'` is returned. The continuation is to be used when matching further objects by calling `match_object/1`.

If the keypos'th element of `Pattern` is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, `N` objects at a time. The default, indicated by giving `N` the value `default`, is to let the number of objects vary depending on the sizes of the objects. If `Name` is a version 9 table, all matching objects with the same key are always returned in the same reply which implies that more than N objects may sometimes be returned.

The table should always be protected using `safe_fixtable/2` before calling `match_object/3`, or errors may occur when calling `match_object/1`.

`member(Name, Key) -> Bool | {error, Reason}`

Types:

- Name = name()
- Key = term()
- Bool = bool()

Works like `lookup/2`, but does not return the objects. The function returns `true` if one or more elements of the table has the key `Key`, `false` otherwise.

`next(Name, Key1) -> Key2 | '$end_of_table'`

Types:

- Name = name()
- Key1 = Key2 = term()

Returns the key following `Key1` in the table `Name` according to the table's internal order, or `'$end_of_table'` if there is no next key.

Should an error occur, the process is exited with an error tuple `{error, Reason}`.

Use `first/1` to find the first key in the table.

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

Types:

- FileName = file()
- Reference = ref()

Opens an existing table. If the table has not been properly closed, the error `{error, need_repair}` is returned. The returned reference is to be used as the name of the table. This function is most useful for debugging purposes.

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

Types:

- Name = atom()

Opens a table. An empty Dets table is created if no file exists.

The atom `Name` is the name of the table. The table name must be provided in all subsequent operations on the table. The name can be used by other processes as well, and several process can share one table.

If two processes open the same table by giving the same name and arguments, then the table will have two users. If one user closes the table, it still remains open until the second user closes the table.

The `Args` argument is a list of `{Key, Val}` tuples where the following values are allowed:

- `{access, access()}`. It is possible to open existing tables in read-only mode. A table which is opened in read-only mode is not subjected to the automatic file reparation algorithm if it is later opened after a crash. The default value is `read_write`.
- `{auto_save, auto_save()}`, the auto save interval. If the interval is an integer `Time`, the table is flushed to disk whenever it is not accessed for `Time` milliseconds. A table that has been flushed will require no reparation when reopened after an uncontrolled emulator halt. If the interval is the atom `infinity`, auto save is disabled. The default value is 180000 (3 minutes).
- `{estimated_no_objects, int()}`. Equivalent to the `min_no_slots` option.
- `{file, file()}`, the name of the file to be opened. The default value is the name of the table.
- `{max_no_slots, no_slots()}`, the maximum number of slots that will be used. The default value is 2 M, and the maximal value is 32 M. Note that a higher value may increase the fragmentation of the table, and conversely, that a smaller value may decrease the fragmentation, at the expense of execution time. Only available for version 9 tables.
- `{min_no_slots, no_slots()}`. Application performance can be enhanced with this flag by specifying, when the table is created, the estimated number of different keys that will be stored in the table. The default value as well as the minimum value is 256.
- `{keypos, keypos()}`, the position of the element of each object to be used as key. The default value is 1. The ability to explicitly state the key position is most convenient when we want to store Erlang records in which the first position of the record is the name of the record type.
- `{ram_file, bool()}`, whether the table is to be kept in RAM. Keeping the table in RAM may sound like an anomaly, but can enhance the performance of applications which open a table, insert a set of objects, and then close the table. When the table is closed, its contents are written to the disk file. The default value is `false`.
- `{repair, Value}`. `Value` can be either a `bool()` or the atom `force`. The flag specifies whether the Dets server should invoke the automatic file reparation algorithm. The default is `true`. If `false` is specified, there is no attempt to repair the file and `{error, need_repair}` is returned if the table needs to be repaired.

  The value `force` means that a reparation will take place even if the table has been properly closed. This is how to convert tables created by older versions of STDLIB. An example is tables hashed with the deprecated `erlang:hash/2` BIF. Tables created with Dets from a STDLIB version of 1.8.2 and later use the `erlang:phash/2` function or the `erlang:phash2/1` function, which is preferred.

  The `repair` option is ignored if the table is already open.

- {`type, type()`}, the type of the table. The default value is `set`.
- {`version, version()`}, the version of the format used for the table. The default value is `9`. Tables on the format used before OTP R8 can be created by giving the value `8`. A version 8 table can be converted to a version 9 table by giving the options {`version,9`} and {`repair,force`}.

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

Types:

- Name = name()
- Pid = pid()

Returns the name of the table given the pid of a process that handles requests to a table, or `undefined` if there is no such table.

This function is meant to be used for debugging only.

`safe_fixtable(Name, Fix)`

Types:

- Name = name()
- Fix = bool()

If `Fix` is `true`, the table `Name` is fixed (once more) by the calling process, otherwise the table is released. The table is also released when a fixing process terminates.

If several processes fix a table, the table will remain fixed until all processes have released it or terminated. A reference counter is kept on a per process basis, and N consecutive fixes require N releases to release the table.

It is not guaranteed that calls to `first/1`, `next/2`, select and match functions work as expected even if the table has been fixed; the limited support for concurrency implemented in Ets has not yet been implementeded in Dets. Fixing a table currently only disables resizing of the hash list of the table.

If objects have been added while the table was fixed, the hash list will start to grow when the table is released which will significantly slow down access to the table for a period of time.

`select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = select_cont()
- Selection = [term()]

Returns the results of applying a match specification to some objects stored in a table. The table, the match specification, and the number of objects that are matched are all defined by `Continuation`, which has been returned by a prior call to `select/1` or `select/3`.

When all objects of the table have been matched, `'$end_of_table'` is returned.

`select(Name, MatchSpec) -> Selection | {error, Reason}`

Types:

- Name = name()

- MatchSpec = match_spec()
- Selection = [term()]

Returns the results of applying the match specification `MatchSpec` to all or some objects stored in the table `Name`. The order of the objects is not specified. See the ERTS User's Guide for a description of match specifications.

If the keypos'th element of `MatchSpec` is unbound, the match specification is applied to all objects of the table. If the keypos'th element is bound, the match specification is applied to the objects with the right key(s) only.

Using the `select` functions for traversing all objects of a table is more efficient than calling `first/1` and `next/2` or `slot/2`.

`select(Name, MatchSpec, N) -> {Selection, Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- MatchSpec = match_spec()
- N = default | int()
- Selection = [term()]
- Continuation = select_cont()

Returns the results of applying the match specification `MatchSpec` to some or all objects stored in the table `Name`. The order of the objects is not specified. See the ERTS User's Guide for a description of match specifications.

A tuple of the results of applying the match specification and a continuation is returned, unless the table is empty, in which case `'$end_of_table'` is returned. The continuation is to be used when matching further objects by calling `select/1`.

If the keypos'th element of `MatchSpec` is bound, the match specification is applied to all objects of the table with the right key(s). If the keypos'th element of `MatchSpec` is unbound, the match specification is applied to all objects of the table, `N` objects at a time. The default, indicated by giving `N` the value `default`, is to let the number of objects vary depending on the sizes of the objects. If `Name` is a version 9 table, all objects with the same key are always handled at the same time which implies that the match specification may be applied to more than N objects.

The table should always be protected using `safe_fixtable/2` before calling `select/3`, or errors may occur when calling `select/1`.

`select_delete(Name, MatchSpec) -> N | {error, Reason}`

Types:

- Name = name()
- MatchSpec = match_spec()
- N = int()

Deletes each object from the table `Name` such that applying the match specification `MatchSpec` to the object returns the value `true`. See the ERTS User's Guide for a description of match specifications. Returns the number of deleted objects.

If the keypos'th element of `MatchSpec` is bound, the match specification is applied to the objects with the right key(s) only.

`slot(Name, I) -> '$end_of_table' | [Object] | {error, Reason}`

Types:

- Name = name()
- I = int()
- Object = object()

The objects of a table are distributed among slots, starting with slot `0` and ending with slot n. This function returns the list of objects associated with slot `I`. If `I` is greater than n `'$end_of_table'` is returned.

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

Types:

- Name = name()

Ensures that all updates made to the table `Name` are written to disk. This also applies to tables which have been opened with the `ram_file` flag set to `true`. In this case, the contents of the RAM file are flushed to disk.

Note that the space management data structures kept in RAM, the buddy system, is also written to the disk. This may take some time if the table is fragmented.

`to_ets(Name, EtsTab) -> EtsTab | {error, Reason}`

Types:

- Name = name()
- EtsTab = -see ets(3)-

Inserts the objects of the Dets table `Name` into the Ets table `EtsTab`. The order in which the objects are inserted is not specified. The existing objects of the Ets table are kept unless overwritten.

`traverse(Name, Fun) -> Return | {error, Reason}`

Types:

- Fun = fun(Object) -> FunReturn
- FunReturn = continue | {continue, Val} | {done, Value}
- Val = Value = term()
- Name = name()
- Object = object()
- Return = [term()]

Applies `Fun` to each object stored in the table `Name` in some unspecified order. 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 can be used to print out the contents of a table:

```
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 of a table in a list:

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

{done, Value} Terminate the traversal and return [Value | Acc].

Any other value returned by Fun terminates the traversal and is immediately returned.

update_counter(Name, Key, Increment) -> Result

Types:

- Name = name()
- Key = term()
- Increment = {Pos, Incr} | Incr
- Pos = Incr = Result = integer()

Updates the object with key Key stored in the table Name of type set by adding Incr to the element at the Pos:th position. The new counter value is returned. If no position is specified, the element directly following the key is updated.

This functions provides a way of updating a counter, without having to look up an object, update the object by incrementing an element and insert the resulting object into the table again.

See Also

ets(3) [page 117], mnesia(3)

dict

Erlang Module

`Dict` implements a `Key` - `Value` dictionary. The representation of a dictionary is not defined.

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.

`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.

`filter(Pred, Dict1) -> Dict2`

Types:

- Pred = fun(Key, Value) -> bool()
- Dict1 = Dict2 = dictionary()

`Dict2` is a dictionary of all keys and values in `Dict1` for which `Pred(Key, Value)` is `true`.

`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.

`fold(Function, Acc0, Dict) -> Acc1`

Types:

- Function = fun(Key, Value, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Dict = dictionary()

Calls `Function` on successive keys and values of `Dict` 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. The evaluation order is undefined.

`from_list(List) -> Dict`

Types:

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

This function converts the key/value list `List` to a dictionary.

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

Types:

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

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

`map(Func, Dict1) -> Dict2`

Types:

- Func = fun(Key, Value) -> Value
- Dict1 = Dict2 = dictionary()

`map` calls `Func` on successive keys and values of Dict to return a new value for each key. The evaluation order is undefined.

`merge(Func, Dict1, Dict2) -> Dict3`

Types:

- Func = fun(Key, Value1, Value2) -> Value
- Dict1 = Dict2 = Dict3 = dictionary()

`merge` merges two dictionaries, Dict1 and Dict2, to create a new dictionary. All the `Key` - `Value` pairs from both dictionaries are included in the new dictionary. If a key occurs in both dictionaries then `Func` is called with the key and both values to return a new value. `merge` could be defined as:

```
merge(Fun, D1, D2) ->
    fold(fun (K, V1, D) ->
                 update(K, fun (V2) -> Fun(K, V1, V2) end, V1, D)
         end, D2, D1).
```

but is faster.

`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`.

`to_list(Dict) -> List`

Types:

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

This function converts the dictionary to a list representation.

`update(Key, Function, Dict) -> Dict`

Types:

- Key = term()
- Function = fun(Value) -> Value
- Dict = dictionary()

Update the a value in a dictionary by calling `Function` on the value to get a new value. An exception is generated if `Key` is not present in the dictionary.

`update(Key, Function, Initial, Dict) -> Dict`

Types:

- Key = Initial = term()
- Function = fun(Value) -> Value
- Dict = dictionary()

Update the a value in a dictionary by calling `Function` on the value to get a new value. If `Key` is not present in the dictionary then `Initial` will be stored as the first value. For example we could define append/3 as:

```
append(Key, Val, D) ->
    update(Key, fun (Old) -> Old ++ [Val] end, [Val], D).
```

`update_counter(Key, Increment, Dict) -> Dict`

Types:

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

Add `Increment` to the value associated with `Key` and store this value. If `Key` is not present in the dictionary then `Increment` will be stored as the first value.

This is could have been defined as:

```
update_counter(Key, Incr, D) ->
    update(Key, fun (Old) -> Old + Incr end, Incr, D).
```

but is faster.

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`.

See Also

gb_trees(3) [page 151], orddict(3) [page 221]

digraph

Erlang Module

The `digraph` module implements a version of labeled directed graphs. What makes the graphs implemented here non-proper directed graphs is that multiple edges between vertices are allowed. However, the customary definition of directed graphs will be used in the text that follows.

A *directed graph* (or just "digraph") is a pair (V,E) of a finite set V of *vertices* and a finite set E of *directed edges* (or just "edges"). The set of edges E is a subset of VV (the Cartesian product of V with itself). In this module, V is allowed to be empty; the so obtained unique digraph is called the *empty digraph*. Both vertices and edges are represented by unique Erlang terms.

Digraphs can be annotated with additional information. Such information may be attached to the vertices and to the edges of the digraph. A digraph which has been annotated is called a *labeled digraph*, and the information attached to a vertex or an edge is called a *label*. Labels are Erlang terms.

An edge e=(v,w) is said to *emanate* from vertex v and to be *incident* on vertex w. The *out-degree* of a vertex is the number of edges emanating from that vertex. The *in-degree* of a vertex is the number of edges incident on that vertex. If there is an edge emanating from v and incident on w, then w is is said to be an *out-neighbour* of v, and v is said to be an *in-neighbour* of w. A *path* P from v[1] to v[k] in a digraph (V, E) is a non-empty sequence v[1],v[2],...,v[k] of vertices in V such that there is an edge (v[i],v[i+1]) in E for 1<=i<k. The *length* of the path P is k-1. P is *simple* if all vertices are distinct, except that the first and the last vertices may be the same. P is a *cycle* if the length of P is not zero and v[1] = v[k]. A *loop* is a cycle of length one. A *simple cycle* is a path that is both a cycle and simple. An *acyclic digraph* is a digraph that has no cycles.

Exports

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

Types:

- G = digraph()
- E = edge()
- V1 = V2 = vertex()
- Label = label()
- Reason = {bad_edge, Path} | {bad_vertex, V}
- Path = [vertex()]

add_edge/5 creates (or modifies) the edge E of the digraph G, using Label as the (new) label [page 81] of the edge. The edge is emanating [page 81] from V1 and incident [page 81] on V2. Returns E.

add_edge(G,V1,V2,Label) is equivalent to add_edge(G,E,V1,V2,Label), where E is a created edge. Tuples on the form ['$e'|N], where N is an integer>=1, are used for representing the created edges.

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

If the edge would create a cycle in an acyclic digraph [page 81], then {error,{bad_edge,Path}} is returned. If either of V1 or V2 is not a vertex of the digraph G, then {error,{bad_vertex,V}} is returned, V=V1 or V=V2.

```
add_vertex(G, V, Label) -> vertex()
add_vertex(G, V) -> vertex()
add_vertex(G) -> vertex()
```

Types:

- G = digraph()
- V = vertex()
- Label = label()

add_vertex/3 creates (or modifies) the vertex V of the digraph G, using Label as the (new) label [page 81] of the vertex. Returns V.

add_vertex(G,V) is equivalent to add_vertex(G,V,[]).

add_vertex/1 creates a vertex using the empty list as label, and returns the created vertex. Tuples on the form ['$v'|N], where N is an integer>=1, are used for representing the created vertices.

```
del_edge(G, E) -> true
```

Types:

- G = digraph()
- E = edge()

Deletes the edge E from the digraph G.

```
del_edges(G, Edges) -> true
```

Types:

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

Deletes the edges in the list Edges from the digraph G.

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

Types:

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

Deletes edges from the digraph `G` until there are no paths [page 81] from the vertex `V1` to the vertex `V2`.

A sketch of the procedure employed: Find an arbitrary simple path [page 81] v[1],v[2],...,v[k] from `V1` to `V2` in `G`. Remove all edges of `G` emanating [page 81] from v[i] and incident [page 81] to v[i+1] for 1<=i<k (including multiple edges). Repeat until there is no path between `V1` and `V2`.

`del_vertex(G, V) -> true`

Types:

- G = digraph()
- V = vertex()

Deletes the vertex `V` from the digraph `G`. Any edges emanating [page 81] from `V` or incident [page 81] on `V` are also deleted.

`del_vertices(G, Vertices) -> true`

Types:

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

Deletes the vertices in the list `Vertices` from the digraph `G`.

`delete(G) -> true`

Types:

- G = digraph()

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

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

Types:

- G = digraph()
- E = edge()
- V1 = V2 = vertex()
- Label = label()

Returns `{E,V1,V2,Label}` where `Label` is the label [page 81] of the edge `E` emanating [page 81] from `V1` and incident [page 81] on `V2` of the digraph `G`. If there is no edge `E` of the digraph `G`, then `false` is returned.

`edges(G) -> Edges`

Types:

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

Returns a list of all edges of the digraph `G`, in some unspecified order.

`edges(G, V) -> Edges`

Types:

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

Returns a list of all edges emanating [page 81] from or incident [page 81] on `V` of the digraph `G`, in some unspecified order.

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

Types:

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

If there is a simple cycle [page 81] of length two or more through the vertex `V`, then the cycle is returned as a list `[V,...,V]` of vertices, otherwise if there is a loop [page 81] through `V`, then the loop is returned as a list `[V]`. If there are no cycles through `V`, then `false` is returned.

`get_path/3` is used for finding a simple cycle through `V`.

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

Types:

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

Tries to find a simple path [page 81] from the vertex `V1` to the vertex `V2` of the digraph `G`. Returns the path as a list `[V1,...,V2]` of vertices, or `false` if no simple path from `V1` to `V2` of length one or more exists.

The digraph `G` is traversed in a depth-first manner, and the first path found is returned.

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

Types:

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

Tries to find an as short as possible simple cycle [page 81] through the vertex `V` of the digraph `G`. Returns the cycle as a list `[V,...,V]` of vertices, or `false` if no simple cycle through `V` exists. Note that a loop [page 81] through `V` is returned as the list `[V,V]`.

`get_short_path/3` is used for finding a simple cycle through `V`.

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

Types:

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

Tries to find an as short as possible simple path [page 81] from the vertex `V1` to the vertex `V2` of the digraph `G`. Returns the path as a list `[V1,...,V2]` of vertices, or `false` if no simple path from `V1` to `V2` of length one or more exists.

The digraph `G` is traversed in a breadth-first manner, and the first path found is returned.

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

Types:

- G= digraph()
- V = vertex()

Returns the in-degree [page 81] of the vertex `V` of the digraph `G`.

`in_edges(G, V) -> Edges`

Types:

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

Returns a list of all edges incident [page 81] on `V` of the digraph `G`, in some unspecified order.

`in_neighbours(G, V) -> Vertices`

Types:

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

Returns a list of all in-neighbours [page 81] of `V` of the digraph `G`, in some unspecified order.

`info(G) -> InfoList`

Types:

- G = digraph()
- InfoList = [{cyclicity, Cyclicity}, {memory, NoWords}, {protection, Protection}]
- Cyclicity = cyclic | acyclic
- Protection = public | protected | private
- NoWords = integer() >= 0

Returns a list of `{Tag, Value}` pairs describing the digraph `G`. The following pairs are returned:

- `{cyclicity, Cyclicity}`, where `Cyclicity` is `cyclic` or `acyclic`, according to the options given to `new`.
- `{memory, NoWords}`, where `NoWords` is the number of words allocated to the `ets` tables.
- `{protection, Protection}`, where `Protection` is `public`, `protected` or `private`, according to the options given to `new`.

`new() -> digraph()`

Equivalent to `new([])`.

`new(Type) -> digraph() | {error, Reason}`

Types:

- Type = [cyclic | acyclic | public | private | protected]
- Reason = {unknown_type, term()}

Returns an empty digraph [page 81] with properties according to the options in `Type`:

`cyclic` Allow cycles [page 81] in the digraph (default).

`acyclic` The digraph is to be kept acyclic [page 81].

`public` The digraph may be read and modified by any process.

`protected` Other processes can only read the digraph (default).

`private` The digraph can be read and modified by the creating process only.

If an unrecognized type option T is given, then `{error,{unknown_type,T}}` is returned.

`no_edges(G) -> integer() >= 0`

Types:

- G = digraph()

Returns the number of edges of the digraph `G`.

`no_vertices(G) -> integer() >= 0`

Types:

- G = digraph()

Returns the number of vertices of the digraph `G`.

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

Types:

- G = digraph()
- V = vertex()

Returns the out-degree [page 81] of the vertex `V` of the digraph `G`.

`out_edges(G, V) -> Edges`

Types:

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

Returns a list of all edges emanating [page 81] from `V` of the digraph `G`, in some unspecified order.

`out_neighbours(G, V) -> Vertices`

Types:

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

Returns a list of all out-neighbours [page 81] of `V` of the digraph `G`, in some unspecified order.

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

Types:

- G = digraph()
- V = vertex()
- Label = label()

Returns `{V,Label}` where `Label` is the label [page 81] of the vertex `V` of the digraph `G`, or `false` if there is no vertex `V` of the digraph `G`.

`vertices(G) -> Vertices`

Types:

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

Returns a list of all vertices of the digraph `G`, in some unspecified order.

See Also

digraph_utils(3) [page 88], ets(3) [page 117]

digraph_utils

Erlang Module

The `digraph_utils` module implements some algorithms based on depth-first traversal of directed graphs. See the `digraph` module for basic functions on directed graphs.

A *directed graph* (or just "digraph") is a pair (V,E) of a finite set V of *vertices* and a finite set E of *directed edges* (or just "edges"). The set of edges E is a subset of VV (the Cartesian product of V with itself).

Digraphs can be annotated with additional information. Such information may be attached to the vertices and to the edges of the digraph. A digraph which has been annotated is called a *labeled digraph*, and the information attached to a vertex or an edge is called a *label*.

An edge e=(v,w) is said to *emanate* from vertex v and to be *incident* on vertex w. If there is an edge emanating from v and incident on w, then w is is said to be an *out-neighbour* of v. A *path* P from v[1] to v[k] in a digraph (V, E) is a non-empty sequence v[1],v[2],...,v[k] of vertices in V such that there is an edge (v[i],v[i+1]) in E for 1<=i<k. The *length* of the path P is k-1. P is a *cycle* if the length of P is not zero and v[1] = v[k]. A *loop* is a cycle of length one. An *acyclic digraph* is a digraph that has no cycles.

A *depth-first traversal* of a directed digraph can be viewed as a process that visits all vertices of the digraph. Initially, all vertices are marked as unvisited. The traversal starts with an arbitrarily chosen vertex, which is marked as visited, and follows an edge to an unmarked vertex, marking that vertex. The search then proceeds from that vertex in the same fashion, until there is no edge leading to an unvisited vertex. At that point the process backtracks, and the traversal continues as long as there are unexamined edges. If there remain unvisited vertices when all edges from the first vertex have been examined, some hitherto unvisited vertex is chosen, and the process is repeated.

A *partial ordering* of a set S is a transitive, antisymmetric and reflexive relation between the objects of S. The problem of *topological sorting* is to find a total ordering of S that is a superset of the partial ordering. A digraph G=(V,E) is equivalent to a relation E on V (we neglect the fact that the version of directed graphs implemented in the `digraph` module allows multiple edges between vertices). If the digraph has no cycles of length two or more, then the reflexive and transitive closure of E is a partial ordering.

A *subgraph* G' of G is a digraph whose vertices and edges form subsets of the vertices and edges of G. G' is *maximal* with respect to a property P if all other subgraphs that include the vertices of G' do not have the property P. A *strongly connected component* is a maximal subgraph such that there is a path between each pair of vertices. A *connected component* is a maximal subgraph such that there is a path between each pair of vertices, considering all edges undirected.

Exports

`components(Digraph) -> [Component]`

Types:

- Digraph = digraph()
- Component = [vertex()]

Returns a list of connected components [page 88]. Each component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Each vertex of the digraph `Digraph`occurs in exactly one component.

`condensation(Digraph) -> CondensedDigraph`

Types:

- Digraph = CondensedDigraph = digraph()

Creates a digraph where the vertices are the strongly connected components [page 88] of `Digraph` as returned by `strong_components/1`. If X and Y are strongly connected components, and there exist vertices x and y in X and Y respectively such that there is an edge emanating [page 88] from x and incident [page 88] on y, then an edge emanating from X and incident on Y is created.

The created digraph has the same type as `Digraph`. All vertices and edges have the default label [page 88] `[]`.

Each and every cycle [page 88] is included in some strongly connected component, which implies that there always exists a topological ordering [page 88] of the created digraph.

`cyclic_strong_components(Digraph) -> [StrongComponent]`

Types:

- Digraph = digraph()
- StrongComponent = [vertex()]

Returns a list of strongly connected components [page 88]. Each strongly component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Only vertices that are included in some cycle [page 88] in `Digraph` are returned, otherwise the returned list is equal to that returned by `strong_components/1`.

`is_acyclic(Digraph) -> bool()`

Types:

- Digraph = digraph()

Returns `true` if and only if the digraph `Digraph` is acyclic [page 88].

`loop_vertices(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns a list of all vertices of `Digraph` that are included in some loop [page 88].

`postorder(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 88] of the digraph, collecting visited vertices in postorder. More precisely, the vertices visited while searching from an arbitrarily chosen vertex are collected in postorder, and all those collected vertices are placed before the subsequently visited vertices.

`preorder(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 88] of the digraph, collecting visited vertices in pre-order.

`reachable(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 88] in `Digraph` from some vertex of `Vertices` to the vertex. In particular, since paths may have length zero, the vertices of `Vertices` are included in the returned list.

`reachable_neighbours(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 88] in `Digraph` of length one or more from some vertex of `Vertices` to the vertex. As a consequence, only those vertices of `Vertices` that are included in some cycle [page 88] are returned.

`reaching(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 88] from the vertex to some vertex of `Vertices`. In particular, since paths may have length zero, the vertices of `Vertices` are included in the returned list.

`reaching_neighbours(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 88] of length one or more from the vertex to some vertex of `Vertices`. As a consequence, only those vertices of `Vertices` that are included in some cycle [page 88] are returned.

`strong_components(Digraph) -> [StrongComponent]`

Types:

- Digraph = digraph()
- StrongComponent = [vertex()]

Returns a list of strongly connected components [page 88]. Each strongly component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Each vertex of the digraph `Digraph` occurs in exactly one strong component.

`subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}`

Types:

- Digraph = Subgraph = digraph()
- Options = [{type, SubgraphType}, {keep_labels, bool()}]
- Reason = {invalid_option, term()} | {unknown_type, term()}
- SubgraphType = inherit | type()
- Vertices = [vertex()]

Creates a maximal subgraph [page 88] of `Digraph` having as vertices those vertices of `Digraph` that are mentioned in `Vertices`.

If the value of the option `type` is `inherit`, which is the default, then the type of `Digraph` is used for the subgraph as well. Otherwise the option value of `type` is used as argument to `digraph:new/1`.

If the value of the option `keep_labels` is `true`, which is the default, then the labels [page 88] of vertices and edges of `Digraph` are used for the subgraph as well. If the value is `false`, then the default label, `[]`, is used for the subgraph's vertices and edges.

`subgraph(Digraph, Vertices)` is equivalent to `subgraph(Digraph, Vertices, [])`.

`topsort(Digraph) -> Vertices | false`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns a topological ordering [page 88] of the vertices of the digraph `Digraph` if such an ordering exists, `false` otherwise. For each vertex in the returned list, there are no out-neighbours [page 88] that occur earlier in the list.

See Also

digraph(3) [page 81]

epp

Erlang 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(3) [page 103]

erl_eval

Erlang 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}`

`exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {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
- NonlocalFunctionHandler = {value, 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 arguments `LocalFunctionHandler` and `NonlocalFunctionHandler`.

Returns `{value, Value, NewBindings}`

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

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

`expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> { 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
- NonlocalFunctionHandler = {value, 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 arguments `LocalFunctionHandler` and `NonlocalFunctionHandler`.

Returns `{value, Value, NewBindings}`.

`expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
`expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
`expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {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 (an atom) 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 (an atom), `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.

Non-local Function Handler

The optional argument NonlocalFunctionHandler may be used to define a function which is called in the following cases: a functional object (fun) is called; a built-in function is called; a function is called using the M:F syntax, where M and F are atoms or expressions. Exceptions are function calls in guard tests and calls to erlang:apply/2,3; neither of the function handlers will be called for such calls. The argument can have the following formats:

{value,Func} This defines an nonlocal function handler which is called with:

```
Func(FuncSpec, Arguments)
```

FuncSpec is the name of the function on the form {Module,Function} or a fun, and Arguments is a list of the *evaluated* arguments. The function handler returns the value of the function. To signal an error, the function handler just calls exit/1 with a suitable exit value.

none There is no nonlocal function handler.

The nonlocal function handler argument is probably not used as frequently as the local function handler argument. A possible use is to call exit/1 on calls to functions that for some reason are not allowed to be called.

Bugs

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

Any undocumented functions in erl_eval should not be used.

erl_id_trans

Erlang 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(3) [page 103], compile(3).

erl_internal

Erlang 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

Erlang 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:

{warn_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 and incorrect number of arguments; 2 = warnings also when the validity could not be checked (for example, when the format string argument is a variable). The default verbosity is 1. Verbosity 0 can also be selected by the option nowarn_format.

warn_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"). No warnings for unused variables, which is the default, can be selected by the option nowarn_unused_vars.

warn_export_vars Causes warnings to be emitted for all implicitly exported variables referred to after the primitives where they were first defined. No warnings for exported variables unless they are referred to in some pattern, which is the default, can be selected by the option nowarn_export_vars.

warn_shadow_vars Causes warnings to be emitted for "fresh" variables in functional objects or list comprehensions with the same name as some already defined variable. The default is to warn for such variables. No warnings for shadowed variables can be selected by the option nowarn_shadow_vars.

warn_unused_import Causes warnings to be emitted for unused imported functions. No warnings for imported functions, which is the default, can be selected by the option nowarn_unused_import.

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(3) [page 92], or parser erl_parse(3) [page 103] 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(3) [page 103], epp(3) [page 92]

erl_parse

Erlang 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. The Abstract Format is described in the ERTS User's Guide. 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. `AbsForm` is the abstract form of the parsed form.

`{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 abstract forms of the parsed expressions.

`{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`.

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(3) [page 182], erl_scan(3) [page 109], ERTS User's Guide

erl_pp

Erlang 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(3) [page 182], erl_parse(3) [page 103], erl_eval(3) [page 94]

erl_scan

Erlang 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(3) [page 182], erl_parse(3) [page 103]

erl_tar

Erlang Module

The `erl_tar` module archives and extract files to and from a tar file. The tar file format is the POSIX extended tar file format specified in IEEE Std 1003.1 and ISO/IEC9945-1. That is the same format as used by `tar` program on Solaris, but is not the same as used by the GNU tar program.

By convention, the name of a tar file should end in "`.tar`". To abide to the convention, you'll need to add "`.tar`" yourself to the name.

Tar files can be created in one operation using the create/2 [page 113] or create/3 [page 113] function.

Alternatively, for more control, the open [page 115], add/3,4 [page 112], and close/1 [page 112] functions can be used.

To extract all files from a tar file, use the extract/1 [page 113] function. To extract only some files or to be able to specify some more options, use the extract/2 [page 114] function.

To return a list of the files in a tar file, use either the table/1 [page 115] or table/2 [page 115] function. To print a list of files to the Erlang shell, use either the t/1 [page 115] or tt/1 [page 116] function.

To convert an error term returned from one of the functions above to a readable message, use the format_error/1 [page 114] function.

LIMITATIONS

For maximum compatibility, it is safe to archive files with names up to 100 characters in length. Such tar files can generally be extracted by any `tar` program.

If filenames exceed 100 characters in length, the resulting tar file can only be correctly extracted by a POSIX-compatible `tar` program (such as Solaris `tar`), not by GNU tar.

File have longer names than 256 bytes cannot be stored at all.

The filename of the file a symbolic link points is always limited to 100 characters.

Exports

`add(TarDescriptor, Filename, Options) -> RetValue`

Types:

- TarDescriptor = term()
- Filename = filename()
- Options = [Option]
- Option = dereference|verbose
- RetValue = ok|{error,{Filename,Reason}}
- Reason = term()

The `add/3` function adds a file to a tar file that has been opened for writing by open/1 [page 115].

`dereference` By default, symbolic links will be stored as symbolic links in the tar file. Use the `dereference` option to override the default and store the file that the symbolic link points to into the tar file.

`verbose` Print an informational message about the file being added.

`add(TarDescriptor, Filename, NameInArchive, Options) -> RetValue`

Types:

- TarDescriptor = term()
- Filename = filename()
- NameInArchive = filename()
- Options = [Option]
- Option = dereference|verbose
- RetValue = ok|{error,{Filename,Reason}}
- Reason = term()

The `add/4` function adds a file to a tar file that has been opened for writing by open/1 [page 115]. It accepts the same options as add/3 [page 112].

`NameInArchive` is the name under which the file will be stored in the tar file. That is the name that the file will get when it will be extracted from the tar file.

`close(TarDescriptor)`

Types:

- TarDescriptor = term()

The `close/1` function closes a tar file opened by open/1 [page 115].

`create(Name, FileList) ->RetValue`

Types:

- Name = filename()
- FileList = [filename()]
- RetValue = ok|{error,{Name,Reason}} <V>Reason = term()

The `create/2` function creates a tar file and archives the files whose names are given in `FileList` into it.

`create(Name, FileList, OptionList)`

Types:

- Name = filename()
- FileList = [filename()]
- OptionList = [Option]
- Option = compressed | cooked | dereference | verbose
- RetValue = ok | {error,{Name,Reason}} <V>Reason = term()

The `create/3` function creates a tar file and archives the files whose names are given in `FileList` into it.

The options in `OptionList` modify the defaults as follows.

`compressed` The entire tar file will be compressed, as if it has been run through the `gzip` program. To abide to the convention that a compressed tar file should end in "`.tar.gz`" or "`.tgz`", you'll need to add the appropriate extension yourself.

`cooked` By default, the `open/2` function will open the tar file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the tar file without the `raw` option.

`dereference` By default, symbolic links will be stored as symbolic links in the tar file. Use the `dereference` option to override the default and store the file that the symbolic link points to into the tar file.

`verbose` Print an informational message about each file being added.

`extract(Name) -> RetValue`

Types:

- Name = filename()
- RetValue = ok | {error,{Name,Reason}}
- Reason = term()

The `extract/1` function extracts all files from a tar archive.

If the `Name` argument is given as "`{binary,Binary}`", the contents of the binary is assumed to be a tar archive.

If the `Name` argument is given as "`{file,Fd}`", `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.

Otherwise, `Name` should be a filename.

`extract(Name, OptionList)`

Types:

- Name = filename() | {binary,Binary} | {file,Fd}
- Binary = binary()
- Fd = file_descriptor()
- OptionList = [Option]
- Option = {cwd,Cwd} | {files,FileList} | keep_old_files | verbose
- Cwd = [dirname()]

- FileList = [filename()]
- RetValue = ok|{error,{Name,Reason}}
- Reason = term()

The `extract/2` function extracts files from a tar archive.

If the `Name` argument is given as "{`binary,Binary`}", the contents of the binary is assumed to be a tar archive.

If the `Name` argument is given as "{`file,Fd`}", `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.

Otherwise, `Name` should be a filename.

The following options modify the defaults for the extraction as follows.

{`cwd,Cwd`} Files with relative filenames will by default be extracted to the current working directory. Given the {`cwd,Cwd`} option, the `extract/2` function will extract into the directory `Cwd` instead of to the current working directory.

{`files,FileList`} By default, all files will be extracted from the tar file. Given the {`files,Files`} option, the `extract/2` function will only extract the files whose names are included in `FileList`.

`compressed` Given the `compressed` option, the `extract/2` function will uncompress the file while extracting If the tar file is not actually compressed, the `compressed` will effectively be ignored.

`cooked` By default, the `open/2` function will open the tar file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the tar file without the `raw` option.

`keep_old_files` By default, all existing files with the same name as file in the tar file will be overwritten Given the `keep_old_files` option, the `extract/2` function will not overwrite any existing files.

`verbose` Print an informational message as each file is being extracted.

`format_error(Reason) -> string()`

Types:

- Reason = term()

The `format_error/1` converts an error reason term to a human-readable error message string.

`open(Name, OpenModeList) -> RetValue`

Types:

- Name = filename()
- OpenModeList = [OpenMode]
- Mode = read|write|compressed|cooked
- RetValue = {ok,TarDescriptor}|{error,{Name,Reason}} <V>TarDescriptor = term()
- Reason = term()

The open/2 function opens a tar file.

By convention, the name of a tar file should end in ".tar". To abide to the convention, you'll need to add ".tar" yourself to the name.

Note that there is currently no function for reading from an opened tar file, meaning that opening a tar file for reading is not very useful.

Except for read and write (which are mutually exclusive), the following atoms may be added to OpenModeList:

compressed The entire tar file will be compressed, as if it has been run through the gzip program. To abide to the convention that a compressed tar file should end in ".tar.gz" or ".tgz", you'll need to add the appropriate extension yourself.

cooked By default, the open/2 function will open the tar file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the tar file without the raw option.

Use the add/3,4 [page 112] functions to add one file at the time into an opened tar file. When you are finished adding files, use the close [page 112] function to close the tar file.

> **Warning:**
> The TarDescriptor term is not a file descriptor. You should not rely on the specific contents of the TarDescriptor term, as it may change in future versions as more features are added to the erl_tar module.

table(Name) -> RetValue

Types:

- Name = filename()
- RetValue = {ok,[string()]} | {error,{Name,Reason}}
- Reason = term()

The table/1 function retrieves the names of all files in the tar file Name.

table(Name, Options)

Types:

- Name = filename()

The table/2 function retrieves the names of all files in the tar file Name.

t(Name)

Types:

- Name = filename()

The t/1 function prints the names of all files in the tar file Name to the Erlang shell. (Similar to "tart".)

tt(Name)

Types:

- Name = filename()

The tt/1 function prints names and information about all files in the tar file Name to the Erlang shell. (Similar to "tartv".)

ets

Erlang Module

This module is an interface to the Erlang built-in term storage BIFs. These provide the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data. (In the case of `ordered_set`, see below, access time is proportional to the logarithm of the number of objects stored).

Data is organized as a set of dynamic tables, which can store tuples. Each table is created by a process. When the process terminates, the table is automatically destroyed. Every table has access rights set at creation.

Tables are divided into four different types, `set`, `ordered_set`, `bag` and `duplicate_bag`. A `set` or `ordered_set` table can only have one object associated with each key. A `bag` or `duplicate_bag` can have many objects associated with each key.

The number of tables stored at 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 runtime system (i.e. with the `-env` option to `erl/werl`). The actual limit may be slightly higher than the one specified, but never lower.

Note that there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it will not automatically be destroyed unless the owner process terminates. It can be destroyed explicitly by using `delete/1`.

Some implementation details:

- 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 objects in the table.
- `'$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 below will exit with reason `badarg` if any argument is of the wrong format, or if the table identifier is invalid.
The type `tid()` is used to denote a table identifier. Note that the internal structure of this type is implementation-specific.

Exports

`all() -> [Tab]`

Types:

- Tab = tid() | atom()

Returns a list of all tables at the node. Named tables are given by their names, unnamed tables are given by their table identifiers.

`delete(Tab) -> true`

Types:

- Tab = tid() | atom()

Deletes the entire table `Tab`.

`delete(Tab, Key) -> true`

Types:

- Tab = tid() | atom()
- Key = term()

Deletes all objects with the key `Key` from the table `Tab`.

`delete_all_objects(Tab) -> true`

Types:

- Tab = tid() | atom()

Delete all objects in the ETS table `Tab`. The deletion is atomic.

`delete_object(Tab,Object) -> true`

Types:

- Tab = tid() | atom()
- Object = tuple()

Delete the exact object `Object` from the ETS table, leaving objects with the same key but other differences (useful for type `bag`).

`file2tab(Filename) -> {ok,Tab} | {error,Reason}`

Types:

- Filename = string() | atom()
- Tab = tid() | atom()
- Reason = term()

Reads a file produced by `tab2file/2` and creates the corresponding table `Tab`.

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

Types:

- Tab = tid() | atom()
- Key = term()

Returns the first key `Key` in the table `Tab`. If the table is of the `ordered_set` type, the first key in Erlang term order will be returned. If the table is of any other type, the first key according to the table's internal order will be returned. If the table is empty, `'$end_of_table'` will be returned.

Use `next/2` to find subsequent keys in the table.

`fixtable(Tab, true|false) -> true | false`

Types:

- Tab = tid() | atom()

> **Warning:**
> The function is retained for backwards compatibility only. Use `safe_fixtable/2` instead.

Fixes a table for safe traversal. The function 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. It does not keep track of when and how tables are fixed.

`foldl(Function, Acc0, Tab) -> Acc1`

Types:

- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

`Acc0` is returned if the table is empty. This function is similar to `lists:foldl/3`. The order in which the elements of the table are traversed is unspecified, except for tables of type `ordered_set`, for which they are traversed first to last.

`foldr(Function, Acc0, Tab) -> Acc1`

Types:

- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

`Acc0` is returned if the table is empty. This function is similar to `lists:foldr/3`. The order in which the elements of the table are traversed is unspecified, except for tables of type `ordered_set`, for which they are traversed last to first.

`from_dets(Tab, DetsTab) -> Tab`

Types:

- Tab = tid() | atom()
- DetsTab = atom()

Fills an already created ETS table with the objects in the already opened DETS table named `DetsTab`. The ETS table is emptied before the objects are inserted.

`fun2ms(LiteralFun) -> MatchSpec`

Types:

- LiteralFun = fun() literal
- MatchSpec = term()

Pseudo function that by means of a `parse_transform` translates the *literal* `fun()` typed as parameter in the function call to a match specification as described in the `match_spec` manual of `ERTS` users guide. (with literal I mean that the `fun()` needs to textually be written as the parameter of the function, it cannot be held in a variable which in turn is passed to the function).

The parse transform is implemented in the module `ms_transform` and the source *must* include the file `ms_transform.hrl` in `stdlib` for this pseudo function to work. Failing to include the hrl file in the source will result in a runtime error, not a compile time dito. The include file is easiest included by adding the line `-include_lib("stdlib/include/ms_transform.hrl").` to the source file.

The `fun()` is very restricted, it can take only a single parameter (the object to match), a sole variable or a tuple. It needs to use the `is_XXX` guard tests and one cannot use language constructs that have no representation in a match_spec (like `if`, `case`, `recieve` etc). The return value from the fun will be the return value of the resulting match_spec.

Example:

```
2> ets:fun2ms(fun({M,N}) when N > 3 -> M end).
[{{'$1','$2'},[{'>','$2',3}],['$1']}]
```

Variables from the environment can be imported, so that this works:

```
2> X=3.
3
3> ets:fun2ms(fun({M,N}) when N > X -> M end).
[{{'$1','$2'},[{'>','$2',{const,3}}],['$1']}]
```

The imported variables will be replaced by match_spec `const` expressions, which is consistent with the static scoping for erlang `fun()`'s. Local or global function calls can not be in the guard or body of the fun however. Calls to builtin match_spec functions of course is allowed:

```
4> ets:fun2ms(fun({M,N}) when N > X, is_atomm(M) -> M end).
Error: fun containing local erlang function calls
('is_atomm' called in guard) cannot be translated into match_spec
{error,transform_error}
5> ets:fun2ms(fun({M,N}) when N > X, is_atom(M) -> M end).
[{{'$1','$2'},[{'>','$2',{const,3}},{is_atom,'$1'}],['$1']}]
```

As you can see by the example, the function can be called from the shell too. The `fun()` needs to be literally in the call when used from the shell as well. Other means than the parse_transform are used in the shell case, but more or less the same restrictions apply (the exception beeing records, as they are not handled by the shell).

> **Warning:**
> If the parse_transform is not applied to a module which calls this pseudo function, the call will fail in runtime (with a `badarg`). The module `ets` actually exports a function with this name, but it should never relly be called except for when using the function in the shell. If the `parse_transform` is properly applied by including the `ms_transform.hrl` header file, compiled code will never call the function, but the function call is replaced by a literal match_spec.

More information is provided by the `ms_transform` manual page in `stdlib`.

`i() -> void()`

Displays information about all ETS tables on tty.

`i(Tab) -> void()`

Types:

- Tab = tid() | atom()

Browses the table `Tab` on tty.

`info(Tab) -> [{Item,Value}] | undefined`

Types:

- Tab = tid() | atom()
- Item, Value - see below

Returns information about the table `Tab` as a list of `{Item,Value}` tuples:

- `Item=memory, Value=int()`
  The number of words allocated to the table.
- `Item=owner, Value=pid()`
  The pid of the owner of the table.
- `Item=name, Value=atom()`
  The name of the table.
- `Item=size, Value=int()`
  The number of objects inserted in the table.
- `Item=node, Value=atom()`
  The node where the table is stored. This field is no longer meaningful as tables cannot be accessed from other nodes.
- `Item=named_table, Value=true|false`
  Indicates if the table is named or not.
- `Item=type, Value=set|ordered_set|bag|duplicate_bag`
  The table type.
- `Item=keypos, Value=int()`
  The key position.
- `Item=protection, Value=public|protected|private`
  The table access rights.

`info(Tab, Item) -> Value | undefined`

Types:

- Tab = tid() | atom()
- Item, Value - see below

Returns the information associated with `Item` for the table `Tab`. In addition to the {`Item,Value`} pairs defined for `info/1`, the following items are allowed:

- `Item=fixed, Value=true|false`
  Indicates if the table is fixed by any process or not.
- `Item=safe_fixed, Value={FirstFixed,Info}|false`
  If the table has been fixed using `safe_fixtable/2`, the call returns a tuple where `FirstFixed` is the time when the table was first fixed by a process, which may or may not be one of the processes it is fixed by right now.

  `Info` is a possibly empty lists of tuples {`Pid,RefCount`}, one tuple for every process the table is fixed by right now. `RefCount` is the value of the reference counter, keeping track of how many times the table has been fixed by the process.

  If the table never has been fixed, the call returns `false`.

`init_table(Name, InitFun) -> true`

Types:

- Name = atom()
- InitFun = fun(Arg) -> Res
- Arg = read | close
- Res = end_of_input | {[object()], InitFun} | term()

Replaces the existing objects of the table `Tab` with objects created by calling the input function `InitFun`, see below. This function is provided for compatibility with the DETS module, it's not more efficient than filling a table by using `ets:insert/2`.

When called with the argument `read` the function `InitFun` is assumed to return `end_of_input` when there is no more input, or {`Objects, Fun`}, where `Objects` is a list of objects and `Fun` is a new input function. Any other value Value is returned as an error {`error, `{`init_fun, Value`}}. Each input function will be called exactly once, and should an error occur, the last function is called with the argument `close`, the reply of which is ignored.

If the type of the table is `set` and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. This holds also for duplicated objects stored in tables of type `duplicate_bag`.

`insert(Tab, ObjectOrObjects) -> true`

Types:

- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

Inserts the object or all of the objects in the list `ObjectOrObjects` into the table `Tab`. If there already exists an object with the same key as one of the objects, and the table is a `set` or `ordered_set` table, the old object will be replaced. If the list contains more than one object with the same key and the table is a `set/ordered_set`, one will be inserted, which one is not defined.

`insert_new(Tab, ObjectOrObjects) -> bool()`

Types:

- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

This function works exactly like insert/2, with the exception that instead of overwriting objects with the same key (in the case of sets or ordered_sets) or adding more objects with keys already existing in the table (in the case of bags and duplicate_bags), it simly returns `false`. If `ObjectOrObjects` is a list, the function checks *every* key prior to inseting anything. Nothing will be inserted if not *all* keys present in the list are absent from the table.

`last(Tab) -> Key | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key = term()

Returns the last key `Key` according to Erlang term order in the table `Tab` of the `ordered_set` type. If the table is of any other type, the function is synonymous to `first/2`. If the table is empty, `'$end_of_table'` is returned.

Use `prev/2` to find preceding keys in the table.

`lookup(Tab, Key) -> [Object]`

Types:

- Tab = tid() | atom()
- Key = term()
- Object = tuple()

Returns a list of all objects with the key `Key` in the table `Tab`.

If the table is of type `set` or `ordered_set`, the function returns either the empty list or a list with one element, as there cannot be more than one object with the same key. If the table is of type `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Note that the time order of object insertions is preserved; The first object inserted with the given key will be first in the resulting list, and so on.

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, time is proportional to the (binary) logarithm of the number of objects.

`lookup_element(Tab, Key, Pos) -> Elem`

Types:

- Tab = tid() | atom()
- Key = term()

- Pos = int()
- Elem = term() | [term()]

If the table Tab is of type set or ordered_set, the function returns the Pos:th element of the object with the key Key.

If the table is of type bag or duplicate_bag, the functions returns a list with the Pos:th element of every object with the key Key.

If no object with the key Key exists, the function will exit with reason badarg.

match(Tab, Pattern) -> [Match]

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]

Matches the objects in the table Tab against the pattern Pattern.

A pattern is a term that may contain:

- bound parts (Erlang terms),
- '_' which matches any Erlang term, and
- pattern variables: '$N' where N=0,1,...

The function returns a list with one element for each matching object, where each element is an ordered list of pattern variable bindings. An example:

```
> ets:match(T, '$1'). % Matches every object in the table
[{rufsen,dog,7},{brunte,horse,5},{ludde,dog,5}]
> ets:match(T, {'_',dog,'$1'}).
[[7],[5]]
> ets:match(T, {'_',cow,'$1'}).
[]
```

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]
- Continuation = term()

Works like `ets:match/2` but only returns a limited (`Limit`) number of matching objects. The `Continuation` term can then be used in subsequent calls to `ets:match/1` to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using `ets:first/1` and `ets:next/1`.

`'$end_of_table'` is returned if the table is empty.

`match(Continuation) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Match = [term()]
- Continuation = term()

Continues a match started with `ets:match/3`. The next chunk of the size given in the initial `ets:match/3` call is returned together with a new `Continuation` that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

`match_delete(Tab, Pattern) -> true`

Types:

- Tab = tid() | atom()
- Pattern = tuple()

Deletes all objects which match the pattern `Pattern` from the table `Tab`. See `match/2` for a description of patterns.

`match_object(Tab, Pattern) -> [Object]`

Types:

- Tab = tid() | atom()
- Pattern = Object = tuple()

Matches the objects in the table `Tab` against the pattern `Pattern`. See `match/2` for a description of patterns. The function returns a list of all objects which match the pattern.

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the `ordered_set` type, the result is in the same order as in a `first/next` traversal.

`match_object(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]
- Continuation = term()

Works like `ets:match_object/2` but only returns a limited (`Limit`) number of matching objects. The `Continuation` term can then be used in subsequent calls to `ets:match_object/1` to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using `ets:first/1` and `ets:next/1`.

`'$end_of_table'` is returned if the table is empty.

`match_object(Continuation) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Match = [term()]
- Continuation = term()

Continues a match started with `ets:match_object/3`. The next chunk of the size given in the initial `ets:match_object/3` call is returned together with a new `Continuation` that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

`member(Tab, Key) -> true | false`

Types:

- Tab = tid() | atom()
- Key = term()

Works like lookup/2, but does not return the objects. The function returns `true` if one or more elements in the table has the key `Key`, `false` otherwise.

`new(Name, Options) -> tid()`

Types:

- Name = atom()
- Options = [Option]
- Option = Type | Access | named_table | {keypos,Pos}
- Type = set | ordered_set | bag | duplicate_bag
- Access = public | protected | private
- Pos = int()

Creates a new table and returns a table identifier which can be used in subsequent operations. The table identifier can be sent to other processes so that a table can be shared between different processes within a node.

The parameter `Options` is a list of atoms which specifies table type, access rights, key position and if the table is named or not. If one or more options are left out, the default values are used. This means that not specifying any options (`[]`) is the same as specifying `[set,protected,{keypos,1}]`.

- `set` The table is a `set` table - one key, one object, no order among objects. This is the default table type.
- `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 have a somewhat different behavior in some situations than tables of the other types.

- bag The table is a bag table which can have many objects, but only one instance of each object, per key.
- duplicate_bag The table is a duplicate_bag table which can have many objects, including multiple copies of the same object, per key.
- public Any process may read or write to the table.
- protected The owner process can read and write to the table. Other processes can only read the table. This is the default setting for the access rights.
- private Only the owner process can read or write to the table.
- named_table If this option is present, the name Name is associated with the table identifier. The name can then be used instead of the table identifier in subsequent operations.
- {keypos,Pos} Specfies which element in the stored tuples should be used as key. By default, it is the first element, i.e. Pos=1. However, this is not always appropriate. In particular, we do not want the first element to be the key if we want to store Erlang records in a table.

  Note that any tuple stored in the table must have at least Pos number of elements.

`next(Tab, Key1) -> Key2 | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key1 = Key2 = term()

Returns the next key Key2, following the key Key1 in the table Tab. If the table is of the ordered_set type, the next key in Erlang term order is returned. If the table is of any other type, the next key according to the table's internal order is returned. If there is no next key, '$end_of_table' is returned.

Use first/1 to find the first key in the table.

Unless a table of type set, bag or duplicate_bag is protected using safe_fixtable/2, see below, a traversal may fail if concurrent updates are made to the table. If the table is of type ordered_set, the function returns the next key in order, even if the object does no longer exist.

`prev(Tab, Key1) -> Key2 | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key1 = Key2 = term()

Returns the previous key Key2, preceding the key Key1 according the Erlang term order in the table Tab of the ordered_set type. If the table is of any other type, the function is synonymous to next/2. If there is no previous key, '$end_of_table' is returned.

Use last/1 to find the last key in the table.

`rename(Tab, Name) -> Name`

Types:

- Tab = Name = atom()

Renames the named table `Tab` to the new name `Name`. Afterwards, the old name can not be used to access the table. Renaming an unnamed table has no effect.

`safe_fixtable(Tab, true|false) -> true | false`

Types:

- Tab = tid() | atom()

Fixes a table of the `set`, `bag` or `duplicate_bag` table type for safe traversal.

A process fixes a table by calling `safe_fixtable(Tab,true)`. The table remains fixed until the process releases it by calling `safe_fixtable(Tab,false)`, or until the process terminates.

If several processes fix a table, the table will remain fixed until all processes have released it (or terminated). A reference counter is kept on a per process basis, and N consecutive fixes 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 even if objects are removed during the traversal. An example:

```
clean_all_with_value(Tab,X) ->
  safe_fixtable(Tab,true),
  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);
    _ ->
       true
  end,
  clean_all_with_value(Tab,X,ets:next(Tab,Key)).
```

Note that no deleted objects are actually removed from a fixed table until it has been released. If a process fixes a table but 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.

Use `info/2` to retrieve information about which processes have fixed which tables. A system with a lot of processes fixing tables may need a monitor which sends alarms when tables have been fixed for too long.

Note that for tables of the `ordered_set` type, `safe_fixtable/2` is not necessary as calls to `first/1` and `next/2` will always succeed.

`select(Tab, MatchSpec) -> [Object]`

Types:

- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = term()

Matches the objects in the table `Tab` using a match_spec as described in ERTS users guide. This is a more general call than the `ets:match/2` and `ets:match_object/2` calls. In its simplest forms the match_spec's look like this:

- MatchSpec = [MatchFunction]
- MatchFunction = {MatchHead, [Guard], [Result]}
- MatchHead = "Pattern as in ets:match"
- Guard = {"Guardtest name", ...}
- Result = "Term construct"

This means that the match_spec is always a list of one or more tuples (of arity 3). The tuples first element should be a pattern as destcribed in the documentation of `ets:match/2`. The second element of the tuple should be a list of 0 or more guard tests (described below). The third element of the tuple should be a list containing a description of the value to actually return. In almost all normal cases the list contains exactly one term which fully describes the value to return for each object.

The return value is constructed using the "match variables" bound in the MatchHead or using the special match variables `'$_'` (the whole matching object) and `'$$' (all match variables in a list), so that the following <c>ets:match/2` expression:

```
ets:match(Tab,{'$1','$2','$3'})
```

is exactly equivalent to:

```
ets:select(Tab,[{{'$1','$2','$3'},[],['$$']}])
```

- and the following `ets:match_object/2` call:

```
ets:match_object(Tab,{'$1','$2','$1'})
```

is exactly equivalent to

```
ets:select(Tab,[{{'$1','$2','$1'},[],['$_']}])
```

Composite terms can be constructed in the `Result` part either by simply writing a list, so that this code:

```
ets:select(Tab,[{{'$1','$2','$3'},[],['$$']}])
```

gives the same output as:

```
ets:select(Tab,[{{'$1','$2','$3'},[],[['$1','$2','$3']]}])
```

i.e. all the bound variables in the match head as a list. If tuples are to be constructed, one has to write a tuple of arity 1 with the single element in the tuple beeing the tuple one wants to construct (as an ordinary tuple could be mistaken for a `Guard`). Therefore the following call:

```
ets:select(Tab,[{{'$1','$2','$1'},[],['$_']}])
```

gives the same output as:

```
ets:select(Tab,[{{'$1','$2','$1'},[],[{{'$1','$2','$3'}}]}])
```

- this syntax is equivalent to the syntax used in the trace patterns (see the `dbg` module in the `runtime_tools` application).

The `Guard`'s are constructed as tuples where the first element is the name of the test (again, see the `match_spec` documentation in ERTS users guide) and the rest of the elements are the parameters of the test. To check for a specific type (say a list) of the element bound to the match variable `'$1'`, one would write the test as `{is_list, '$1'}`. If the test fails, the object in the table won't match and the next `MatchFunction` (if any) will be tried. Most guard tests present in erlang can be used, but only the new versions prefixed `is_` are allowed (like `is_float`, `is_atom` etc). An exact list of the allowed guard tests is present in the `match_spec` section of ERTS users guide.

The `Guard` section can also contain logic and arithmetic operations, which are written with the same syntax as the guard tests (prefix notation), so that a guard test written in erlang looking like this:

```
is_integer(X), is_integer(Y), X + Y < 4711
```

is expressed like this (X replaced with '$1' and Y with '$2'):

```
[{is_integer, '$1'}, {is_integer, '$2'}, {'<', {'+', '$1',
'$2'}, 4711}]
```

A complete list of the operators is present in the match_spec section of ERTS users guide.

`select(Tab, MatchSpec, Limit) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = term()
- Continuation = term()

Works like `ets:select/2` but only returns a limited (`Limit`) number of matching objects. The `Continuation` term can then be used in subsequent calls to `ets:select/1` to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using `ets:first/1` and `ets:next/1`.

`'$end_of_table'` is returned if the table is empty.

`select(Continuation) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Match = [term()]
- Continuation = term()

Continues a match started with `ets:select/3`. The next chunk of the size given in the initial `ets:select/3` call is returned together with a new `Continuation` that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

`select_delete(Tab, MatchSpec) -> NumDeleted`

Types:

- Tab = tid() | atom()

- Object = tuple()
- MatchSpec = term()
- NumDeleted = integer()

Matches the objects in the table `Tab` using a match_spec as described in ERTS users guide. If the match_spec returns the atom `true` for an object, that object is removed from the table. For any other result from the match_spec the object is retained. This is a more general call than the `ets:match_delete/2` call.

The function returns the number of objects actually deleted from the table.

`select_count(Tab, MatchSpec) -> NumMatched`

Types:

- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = term()
- NumMatched = integer()

Matches the objects in the table `Tab` using a match_spec as described in ERTS users guide. If the match_spec returns the atom `true` for an object, that object considered a match and is counted. For any other result from the match_spec the object is not considered a match and is therefore not counted.

The function could be described as a match_delete/2 that does not actually delete any elements, but only counts them.

The function returns the number of objects matched.

`slot(Tab, I) -> [Object] | '$end_of_table'`

Types:

- Tab = tid() | atom()
- I = int()
- Object = tuple()

This function is mostly for debugging purpouses, Normally one should use `first/next` or `last/prev` instead.

Returns all objects in the `I`:th slot of the table `Tab`. A table can be traversed by repeatedly calling the function, starting with the first slot `I=0` and ending when `'$end_of_table'` is returned. The function will fail with reason `badarg` if the `I` argument is out of range.

Unless a table of type `set`, `bag` or `duplicate_bag` is protected using `safe_fixtable/2`, see above, a traversal may fail if concurrent updates are made to the table. If the table is of type `ordered_set`, the function returns a list containing the `I`:th object in Erlang term order.

`tab2file(Tab, Filename) -> ok | {error,Reason}`

Types:

- Tab = tid() | atom()
- Filename = string() | atom()
- Reason = term()

Dumps the table `Tab` to the file `Filename`. The implementation of this function is not efficient.

`tab2list(Tab) -> [Object]`

Types:

- Tab = tid() | atom()
- Object = tuple()

Returns a list of all objects in the table `Tab`.

`test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`

Types:

- Tuple = tuple()
- MatchSpec = term()
- Result = term()
- Errors = [{warning|error, string()}]

This function is a utility to test the `match_spec`'s used in calls to `ets:select/2`. The function both tests the `MatchSpec` for "syntactic" correctness and runs the match_spec against the object `Tuple`. If the match_spec contains errors, the tuple `{error, Errors}` is returned where `Errors` is a list of natural language descriptions of what was wrong with the match_spec. If the match_spec is syntactically OK, the function returns `{ok,Term}` where `Term` is what would have been the result in a real `ets:select/2` call or `false` if the `match_spec` does not match the object `Tuple`.

This is a useful debugging and test tool, especially when writing complicated `ets:select/2` calls.

`to_dets(Tab, DetsTab) -> Tab`

Types:

- Tab = tid() | atom()
- DetsTab = atom()

Fills an already created/opened DETS table with the objects in the already opened ETS table named `Tab`. The DETS table is emptied before the objects are inserted.

`update_counter(Tab, Key, {Pos,Incr,Threshold,SetValue}) -> Result`
`update_counter(Tab, Key, {Pos,Incr}) -> Result`
`update_counter(Tab, Key, Incr) -> Result`

Types:

- Tab = tid() | atom()
- Key = term()
- Pos = Incr = Threshold = SetValue = Result = int()

This functions provides an efficient way to update a counter, without the hassle of having to look up an object, update the object by incrementing an element and insert the resulting object into the table again.

It will destructively update the object with key `Key` in the table `Tab` by adding `Incr` to the element at the `Pos`:th position. The new counter value is returned. If no position is specified, the element directly following the key (<`keypos`>`+1`) is updated.

If a `Threshold` is specified, the counter will be reset to the value `SetValue` if the following conditions occur:

- The `Incr` is not negative (`>= 0`) and the result would be greater than (>) `Threshold`
- The `Incr` is negative (`< 0`) and the result would be less than (<) `Threshold`

The function will fail with reason `badarg` if:

- the table is not of type `set` or `ordered_set`,
- no object with the right key exists,
- the object has the wrong arity,
- the element to update is not an integer, or,
- any of `Pos`, `Incr`, `Threshold` or `SetValue` is not an integer

file_sorter

Erlang Module

The functions of this module sort terms on files, merge already sorted files, and check files for sortedness. Chunks containing binary terms are read from a sequence of files, sorted internally in memory and written on temporary files, which are merged producing one sorted file as output. Merging is provided as an optimization; it is faster when the files are already sorted, but it always works to sort instead of merge.

On a file, a term is represented by a header and a binary. Two options define the format of terms on files:

- {header, HeaderLength}. HeaderLength determines the number of bytes preceding each binary and containing the length of the binary in bytes. Default is 4. The order of the header bytes is defined as follows: if B is a binary containing a header only, the size Size of the binary is calculated as `<<Size:HeaderLength/unit:8>> = B`.
- {format, Format}. The format determines the function that is applied to binaries in order to create the terms that will be sorted. The default value is binary_term, which is equivalent to funbinary_to_term/1. The value binary is equivalent to `fun(X) -> X end`, which means that the binaries will be sorted as they are. This is the fastest format. If Format is term, io:read/2 is called to read terms. In that case only the default value of the header option is allowed. The format option also determines what is written to the sorted output file: if Format is term then io:format/3 is called to write each term, otherwise the binary prefixed by a header is written. Note that the binary written is the same binary that was read; the results of applying the Format function are thrown away as soon as the terms have been sorted. Reading and writing terms using the io module is very much slower than reading and writing binaries.

Other options are:

- {order, Order}. The default is to sort terms in ascending order, but that can be changed by the value descending or by giving an ordering function Fun. Fun(A,B) should return true if A comes before B in the ordering, false otherwise. Using an ordering function will slow down the sort considerably. The keysort, keymerge and keycheck functions do not accept ordering functions.
- {unique, bool()}. When sorting or merging files, only the first of a sequence of terms that compare equal is output if this option is set to true. The default value is false which implies that all terms that compare equal are output. When checking files for sortedness, a check that no pair of consecutive terms compares equal is done if this option is set to true.

- {tmpdir, TempDirectory}. The directory where temporary files are put can be chosen explicitly. The default, implied by the value "", is to put temporary files on the same directory as the sorted output file. If output is a function (see below), the directory returned by file:get_cwd() is used instead. The names of temporary files are derived from the Erlang nodename (node()), the process identifier of the current Erlang emulator (os:getpid()), and a timestamp (erlang:now()); a typical name would be fs_mynode@myhost_1763_1043_337000_266005.17, where 17 is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.
- {compressed, bool()}. Temporary files and the output file may be compressed. The default value false implies that written files are not compressed. Regardless of the value of the compressed option, compressed files can always be read. Note that reading and writing compressed files is significantly slower than reading and writing uncompressed files.
- {size, Size}. By default approximately 512*1024 bytes read from files are sorted internally. This option should rarely be needed.
- {no_files, NoFiles}. By default 16 files are merged at a time. This option should rarely be needed.

To summarize, here is the syntax of the options:

- Options = [Option] | Option
- Option = {header, HeaderLength} | {format, Format} | {order, Order} | {unique, bool()} | {tmpdir, TempDirectory} | {compressed, bool()} | {size, Size} | {no_files, NoFiles}
- HeaderLength = int() > 0
- Format = binary_term | term | binary | FormatFun
- FormatFun = fun(Binary) -> Term
- Order = ascending | descending | OrderFun
- OrderFun = fun(Term, Term) -> bool()
- TempDirectory = "" | file_name()
- Size = int() > 0
- NoFiles = int() > 1

As an alternative to sorting files, a function of one argument can be given as input. When called with the argument read the function is assumed to return end_of_input or {end_of_input, Value}} when there is no more input (Value is explained below), or {Objects, Fun}, where Objects is a list of binaries or terms depending on the format and Fun is a new input function. Any other value is immediately returned as value of the current call to sort or keysort. Each input function will be called exactly once, and should an error occur, the last function is called with the argument close, the reply of which is ignored.

A function of one argument can be given as output. The results of sorting or merging the input is collected in a non-empty sequence of variable length lists of binaries or terms depending on the format. The output function is called with one list at a time, and is assumed to return a new output function. Any other return value is immediately returned as value of the current call to the sort or merge function. Each output function is called exactly once. When some output function has been applied to all of the results or an error occurs, the last function is called with the argument close, and the reply is returned as value of the current call to the sort or merge function. If a function is given as input and the last input function returns {end_of_input, Value}, the function given as output will be called with the argument {value, Value}. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

```
sort(Log) ->
    {ok, _} = disk_log:open([{name,Log}, {mode,read_only}]),
    Input = input(Log, start),
    Output = output([]),
    Reply = file_sorter:sort(Input, Output, {format,term}),
    ok = disk_log:close(Log),
    Reply.

input(Log, Cont) ->
    fun(close) ->
            ok;
       (read) ->
            case disk_log:chunk(Log, Cont) of
                {error, Reason} ->
                    {error, Reason};
                {Cont2, Terms} ->
                    {Terms, input(Log, Cont2)};
                {Cont2, Terms, _Badbytes} ->
                    {Terms, input(Log, Cont2)};
                eof ->
                    end_of_input
            end
    end.

output(L) ->
    fun(close) ->
            lists:append(lists:reverse(L));
       (Terms) ->
            output([Terms | L])
    end.
```

Further examples of functions as input and output can be found at the end of the file_sorter module; the term format is implemented with functions.

The possible values of Reason returned when an error occurs are:

- bad_object, {bad_object, FileName}. Applying the format function failed for some binary, or the key(s) could not be extracted from some term.
- {bad_term, FileName}. io:read/2 failed to read some term.
- {file_error, FileName, Reason2}. See file(3) for an explanation of Reason2.
- {premature_eof, FileName}. End-of-file was encountered inside some binary term.
- {not_a_directory, FileName}. The file supplied with the tmpdir option is not a directory.

Types

```
Binary = binary()
FileName = file_name()
FileNames = [FileName]
ICommand = read | close
IReply = end_of_input | {end_of_input, Value} | {[Object], Infun} | InputReply
Infun = fun(ICommand) -> IReply
Input = FileNames | Infun
InputReply = Term
KeyPos = int() > 0 | [int() > 0]
OCommand = {value, Value} | [Object] | close
OReply = Outfun | OutputReply
Object = Term | Binary
Outfun = fun(OCommand) -> OReply
Output = FileName | Outfun
OutputReply = Term
Term = term()
Value = Term
```

Exports

`sort(FileName) -> Reply`
`sort(Input, Output) -> Reply`
`sort(Input, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts terms on files.

`sort(FileName)` is equivalent to `sort([FileName], FileName)`.

`sort(Input, Output)` is equivalent to `sort(Input, Output, [])`.

`keysort(KeyPos, FileName) -> Reply`
`keysort(KeyPos, Input, Output) -> Reply`
`keysort(KeyPos, Input, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts tuples on files. The sort is performed on the element(s) mentioned in `KeyPos`. If two tuples compare equal on one element, next element according to `KeyPos` is compared. The sort is stable.

`keysort(N, FileName)` is equivalent to `keysort(N, [FileName], FileName)`.

`keysort(N, Input, Output)` is equivalent to `keysort(N, Input, Output, [])`.

`merge(FileNames, Output) -> Reply`
`merge(FileNames, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | OutputReply

Merges terms on files. Each input file is assumed to be sorted.

`merge(FileNames, Output)` is equivalent to `merge(FileNames, Output, [])`.

`keymerge(KeyPos, FileNames, Output) -> Reply`
`keymerge(KeyPos, FileNames, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | OutputReply

Merges tuples on files. Each input file is assumed to be sorted on key(s).

`keymerge(KeyPos, FileNames, Output)` is equivalent to `keymerge(KeyPos, FileNames, Output, [])`.

`check(FileName) -> Reply`
`check(FileNames, Options) -> Reply`

Types:

- Reply = {ok, [Result]} | {error, Reason}
- Result = {FileName, TermPosition, Term}
- TermPosition = int() > 1

Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

`check(FileName)` is equivalent to `check([FileName], [])`.

`keycheck(KeyPos, FileName) -> CheckReply`
`keycheck(KeyPos, FileNames, Options) -> Reply`

Types:

- Reply = {ok, [Result]} | {error, Reason}
- Result = {FileName, TermPosition, Term}
- TermPosition = int() > 1

Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

`keycheck(KeyPos, FileName)` is equivalent to `keycheck(KeyPos, [FileName], [])`.

filelib

Erlang Module

This module contains utilities on a higher level than the `file` module.

Exports

`ensure_dir(Name) -> true`

Types:

- Name = filename() | dirname()

The `ensure_dir/1` functions checks that all parent directories for the given file or directory name exist, creating them if not.

`file_size(Filename) -> integer()`

The `file_size` function returns the size of the given file.

`fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`

Types:

- Dir = dirname()
- RegExp = regexp()
- Recursive = true|false
- Fun = fun(F, AccIn) -> AccOut
- AccIn = AccOut = term()

The `fold_files/5` function folds the function `Fun` over all files `F` in the directory `Dir` that match the regular expression `RegExp`. If `Recursive` is true all sub-directories to `Dir` are processed.

`is_dir(Name) -> true | false`

Types:

- Name = filename() | dirname()

The `is_dir/1` function returns `true` if `Name` refers to a directory, and `false` otherwise.

`is_file(Name) -> true | false`

Types:

- Name = filename() | dirname()

The `is_file/1` function returns `true` if `Name` refers to a file or a directory, and `false` otherwise.

`last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}}`

Types:

- Name = filename() | dirname()

The `last_modified/1` function returns the date and time the given file or directory was last modified.

`wildcard(Wildcard) -> list()`

Types:

- Wildcard = filename() | dirname()

The `wildcard/1` function returns a list of all files that match Unix-style wildcard-string `Wildcard`.

The wildcard string looks like an ordinary filename, except that certain “wildcard characters” are interpreted in a special way. The following characters are special:

? Matches one character.

* Matches any number of characters up to the end of the filename, the next dot, or the next slash.

{Item,...} Alternation. Matches one of the alternatives.

Other characters represent themselves. Only filenames that have exactly the same character in the same position will match. (Matching is case-sensitive; i.e. “a” will not match “A”).

Note that multiple “*” characters are allowed (as in Unix wildcards, but opposed to Windows/DOS wildcards).

Examples:

The following examples assume that the current directory is the top of an Erlang/OTP installation.

To find all `.beam` files in all applications, the following line can be used:

```
filelib:wildcard("lib/*/ebin/*.beam").
```

To find either `.erl` or `.hrl` in all applications `src` directories, the following

```
filelib:wildcard("lib/*/src/*.?rl")
```

or the following line

```
filelib:wildcard("lib/*/src/*.{erl,hrl}")
```

can be used.

To find all `.hrl` files in either `src` or `include` directories, use:

```
filelib:wildcard("lib/*/{src,include}/*.hrl").
```

To find all `.erl` or `.hrl` files in either `src` or `include` directories, use:

```
filelib:wildcard("lib/*/{src,include}/*.{erl,hrl}")
```

`wildcard(Wildcard, Cwd) -> list()`

Types:

- Wildcard = filename() | dirname()
- Cwd = dirname()

The `wildcard/2` function works like `wildcard/1`, except that instead of the actual working dirctory, `Cwd` will be used.

filename

Erlang 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.

All functions take either a flat list of characters (a “string”) or an atom, or a deep list of strings and atoms. The `file` module accepts filename in the same form. The return value will always be a flat list of characters.

Types

```
filename = string() | atom() | [filename()]
```

Exports

`absname(Filename) -> Absname`

Types:

- Filename = filename()
- 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 = filename()
- 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.

`absname_join(Directory, Filename) -> Absname`

Types:

- Filename = filename()
- Directory = string()
- Absname = string()

Joins an absolute directory with a relative filename. The same as join/2 but more magical. It also assumes that `Directory` is an absolute directory and `Filename` is a relative filename. Might crash otherwise.

The magic consists in that on some platforms, namely they who have tight restrictions on raw filename length, leading parent directory components in `Filename` are matched against trailing directory components in `Directory` so they can be removed from the result - minimizing it's length.

On the other hand, on platforms that support symbolic links such magic is magically not performed since it would not work.

`basename(Filename)`

Types:

- Filename = filename()

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 = filename()

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 = filename()

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 = filename()

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") -> []
```

`flatten(Filename) -> string()`

Types:

- Filename = filename()

Converts a possibly deep list filename consisting of characters and atoms into the corresponding flat string filename.

`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 = filename()

`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 = filename()
- 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.

gb_sets

Erlang Module

An implementation of ordered sets using Prof. Arne Andersson's General Balanced Trees. This can be much more efficient than using ordered lists, for larger sets, but depends on the application. See notes below for details.

Complexity note

The complexity on set operations is bounded by either O(|S|) or O(|T| * log(|S|)), where S is the largest given set, depending on which is fastest for any particular function call. For operating on sets of almost equal size, this implementation is about 3 times slower than using ordered-list sets directly. For sets of very different sizes, however, this solution can be arbitrarily much faster; in practical cases, often between 10 and 100 times. This implementation is particularly suited for accumulating elements a few at a time, building up a large set (more than 100-200 elements), and repeatedly testing for membership in the current set.

As with normal tree structures, lookup (membership testing), insertion and deletion have logarithmic complexity.

Exports

`empty()`

Returns new, empty set.

Alias: new(), for compatibility with `sets'.

`is_empty(S)`

Returns 'true' if S is an empty set, and 'false' otherwise.

`size(S)`

Returns the number of nodes in the set as an integer. Returns 0 (zero) if the set is empty.

`singleton(X)`

Returns a set containing only the element X.

`is_member(X, S)`

Returns `true' if element X is a member of set S, and `false' otherwise.

Alias: is_element(), for compatibility with `sets'.

`insert(X, S)`

Inserts element X into set S, returns the new set. Assumes that the element is not present in S.

`add(X, S)`

Adds element X to set S, returns the new set. If X is already an element in S, nothing is changed.

Alias: add_element(), for compatibility with `sets'.

`delete(X, S)`

Removes element X from set S, returns new set. Assumes that the element exists in the set.

Alias: del_element(), for compatibility with `sets'.

`delete_any(X, T)`

Removes key X from set S if the key is present in the set, otherwise does nothing; returns new set.

`balance(S)`

Rebalances the tree representation of S. Note that this is rarely necessary, but may be motivated when a large number of elements have been deleted from the tree without further insertions. Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

`union(S1, S2)`

Returns a new set that contains each element that is in either S1 or S2 or both, and no other elements.

`union(Ss)`

Returns a new set that contains each element that is in at least one of the sets in the list Ss, and no other elements.

`intersection(S1, S2)`

Returns a new set that contains each element that is in both S1 and S2, and no other elements.

`intersection(Ss)`

Returns a new set that contains each element that is in all of the sets in the list Ss, and no other elements.

`difference(S1, S2)`

Returns a new set that contains each element in S1 that is not also in S2, and no other elements.

Alias: subtract(), for compatibility with `sets'.

`is_subset(S1, S2)`

Returns `true' if each element in S1 is also a member of S2, and `false' otherwise.

`to_list(S)`

Returns an ordered list of all elements in set S. The list never contains duplicates (of course).

`from_list(List)`

Creates a set containing all elements in List, where List may be unordered and contain duplicates.

`from_ordset(L)`

Turns an ordered-set list L into a set. The list must not contain duplicates.

`smallest(S)`

Returns the smallest element in set S. Assumes that the set S is nonempty.

`largest(S)`

Returns the largest element in set S. Assumes that the set S is nonempty.

`take_smallest(S)`

Returns {X, S1}, where X is the smallest element in set S, and S1 is the set S with element X deleted. Assumes that the set S is nonempty.

`take_largest(S)`

Returns {X, S1}, where X is the largest element in set S, and S1 is the set S with element X deleted. Assumes that the set S is nonempty.

`iterator(S)`

Returns an iterator that can be used for traversing the entries of set S; see `next'. The implementation of this is very efficient; traversing the whole set using `next' is only slightly slower than getting the list of all elements using `to_list' and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

`next(T)`

Returns {X, T1} where X is the smallest element referred to by the iterator T, and T1 is the new iterator to be used for traversing the remaining elements, or the atom `none' if no elements remain.

`filter(P, S)`

Filters set S using predicate function P. Included for compatibility with `sets'.

fold(F, A, S)

Folds function F over set S with A as the initial accumulator. Included for compatibility with `sets'.

is_set(S)

Returns 'true' if S appears to be a set, and 'false' otherwise. Not recommended; included for compatibility with `sets'.

SEE ALSO

gb_trees(3) [page 151], ordsets(3) [page 222], sets(3) [page 247]

gb_trees

Erlang Module

An efficient implementation of Prof. Arne Andersson's General Balanced Trees. These have no storage overhead compared to unbalaced binary trees, and their performance is in general better than AVL trees.

Data structure

Data structure:

```
- {Size, Tree}, where ‘Tree’ is composed of nodes of the form:
  - {Key, Value, Smaller, Bigger}, and the "empty tree" node:
  - nil.
```

There is no attempt to balance trees after deletions. Since deletions don't increase the height of a tree, this should be OK.

Original balance condition *h(T) <= ceil(c * log(|T|))* has been changed to the similar (but not quite equivalent) condition *2 ^ h(T) <= |T| ^ c*. This should also be OK.

Performance is comparable to the AVL trees in the Erlang book (and faster in general due to less overhead); the difference is that deletion works for these trees, but not for the book's trees. Behaviour is logaritmic (as it should be).

Exports

`empty()`

Returns a new, empty tree.

`is_empty(T)`

Returns 'true' if T is an empty tree, and 'false' otherwise.

`size(T)`

Returns the number of nodes in the tree as an integer. Returns 0 (zero) if the tree is empty.

`lookup(X, T)`

Looks up key X in tree T; returns {value, V}, or `none' if the key is not present.

`get(X, T)`

Retrieves the value stored with key X in tree T. Assumes that the key is present in the tree, crashes otherwise.

`insert(X, V, T)`

Inserts key X with value V into tree T; returns the new tree. Assumes that the key is *not* present in the tree, crashes otherwise.

`update(X, V, T)`

Updates key X to value V in tree T; returns the new tree. Assumes that the key is present in the tree.

`enter(X, V, T)`

Inserts key X with value V into tree T if the key is not present in the tree, otherwise updates key X to value V in T. Returns the new tree.

`delete(X, T)`

Removes key X from tree T; returns new tree. Assumes that the key is present in the tree, crashes otherwise.

`delete_any(X, T)`

Removes key X from tree T if the key is present in the tree, otherwise does nothing; returns new tree.

`balance(T)`

Rebalances tree T. Note that this is rarely necessary, but may be motivated when a large number of entries have been deleted from the tree without further insertions. Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

`is_defined(X, T)`

Returns `true' if key X is present in tree T, and `false' otherwise.

`keys(T)`

Returns an ordered list of all keys in tree T.

`values(T)`

Returns the list of values for all keys in tree T, sorted by their corresponding keys. Duplicates are not removed.

`to_list(T)`

Returns an ordered list of {Key, Value} pairs for all keys in tree T.

`from_orddict(L)`

Turns an ordered list L of {Key, Value} pairs into a tree. The list must not contain duplicate keys.

`smallest(T)`

Returns {X, V}, where X is the smallest key in tree T, and V is the value associated with X in T. Assumes that the tree T is nonempty.

`largest(T)`

Returns {X, V}, where X is the largest key in tree T, and V is the value associated with X in T. Assumes that the tree T is nonempty.

`take_smallest(T)`

Returns {X, V, T1}, where X is the smallest key in tree T, V is the value associated with X in T, and T1 is the tree T with key X deleted. Assumes that the tree T is nonempty.

`take_largest(T)`

Returns {X, V, T1}, where X is the largest key in tree T, V is the value associated with X in T, and T1 is the tree T with key X deleted. Assumes that the tree T is nonempty.

`iterator(T)`

Returns an iterator that can be used for traversing the entries of tree T; see `next'. The implementation of this is very efficient; traversing the whole tree using `next' is only slightly slower than getting the list of all elements using `to_list' and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

`next(S)`

Returns {X, V, S1} where X is the smallest key referred to by the iterator S, and S1 is the new iterator to be used for traversing the remaining entries, or the atom `none' if no entries remain.

SEE ALSO

gb_sets(3) [page 147], dict(3) [page 77],

gen_event

Erlang Module

A behaviour module for implementing event handling functionality. The OTP event handling model consists of a generic event manager process with an arbitrary number of event handlers which are added and deleted dynamically.

An event manager implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to *OTP Design Principles* for more information.

Each event handler is implemented as a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_event module                   Callback module
----------------                   ---------------
gen_event:start            ----->  -

gen_event:add_handler
gen_event:add_suphandler   ----->  Module:init/1

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

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

-                          ----->  Module:handle_info/2

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

gen_event:swap_handler
gen_event:swap_sup_handler ----->  Module1:terminate/2
                                   Module2:init/1

gen_event:which_handlers   ----->  -

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

-                          ----->  Module:code_change/3
```

Since each event handler is one callback module, an event manager will have several callback modules which are added and deleted dynamically. Therefore `gen_event` is more tolerant of callback module errors than the other behaviours. If a callback function for an installed event handler fails with `Reason`, or returns a bad value `Term`, the event manager will not fail. It will delete the event handler by calling the callback function

`Module:terminate/2` (see below), giving as argument `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively. No other event handler will be affected.

The `sys` module can be used for debugging an event manager.

Note that an event manager *does* trap exit signals automatically.

Unless otherwise stated, all functions in this module fail if the specified event manager does not exist or if bad arguments are given.

Exports

`start() -> Result`
`start(EventMgrName) -> Result`
`start_link() -> Result`
`start_link(EventMgrName) -> Result`

Types:

- EventMgrName = {local,Name} | {global,Name}
- Name = atom()
- Result = {ok,Pid} | {error,{already_started,Pid}}
- Pid = pid()

Creates an event manager.

An event manager started using `start_link` is linked to the calling process. This function must be used if the event manager is included in a supervision tree. An event manager started using `start` is not linked to the calling process.

If `EventMgrName={local,Name}`, the event manager is registered locally as `Name` using `register/2`. If `EventMgrName={global,Name}`, the event manager is registered globally as `Name` using `global:register_name/2`. If no name is provided, the event manager is not registered.

If the event manager is successfully created the function returns `{ok,Pid}`, where `Pid` is the pid of the event manager. If there already exists a process with the specified `EventMgrName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

`add_handler(EventMgrRef, Handler, Args) -> Result`

Types:

- EventMgr = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = ok | {'EXIT',Reason} | term()
- Reason = term()

Adds a new event handler to the event manager `EventMgrRef`. The event manager will call `Module:init/1` to initiate the event handler and its internal state.

`EventMgrRef` can be:

- the pid,
- Name, if the event manager is locally registered,
- {Name,Node}, if the event manager is locally registered at another node, or
- {global,Name}, if the event manager is globally registered.

Handler is the name of the callback module Module or a tuple {Module,Id}, where Id is any term. The {Module,Id} representation makes it possible to identify a specific event handler when there are several event handlers using the same callback module.

Args is an arbitrary term which is passed as the argument to Module:init/1.

If Module:init/1 returns a correct value, the event manager adds the event handler and this function returns ok. If Module:init/1 fails with Reason or returns an unexpected value Term, the event handler is ignored and this function returns {'EXIT',Reason} or Term, respectively.

`add_sup_handler(EventMgrRef, Handler, Args) -> Result`

Types:

- EventMgr = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = ok | {'EXIT',Reason} | term()
- Reason = term()

Adds a new event handler in the same way as add_handler/3 but will also supervise the connection between the event handler and the calling process.

- If the calling process later terminates with Reason, the event manager will delete the event handler by calling Module:terminate/2 with {stop,Reason} as argument.
- If the event handler later is deleted, the event manager sends a message{gen_event_EXIT,Handler,Reason} to the calling process. Reason is one of the following:
  - normal, if the event handler has been removed due to a call to delete_handler/3, or remove_handler has been returned by a callback function (see below).
  - shutdown, if the event handler has been removed because the event manager is terminating.
  - {swapped,NewHandler,Pid}, if the process Pid has replaced the event handler with another event handler NewHandler using a call to swap_handler/3 or swap_sup_handler/3.
  - a term, if the event handler is removed due to an error. Which term depends on the error.

See add_handler/3 for a description of the arguments and return values.

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

Types:

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

Sends an event notification to the event manager `EventMgrRef`. The event manager will call `Module:handle_event/2` for each installed event handler to handle the event.

`notify` is asynchronous and will return immediately after the event notification has been sent. `sync_notify` is synchronous in the sense that it will return `ok` after the event has been handled by all event handlers.

See `add_handler/3` for a description of `EventMgrRef`.

`Event` is an arbitrary term which is passed as one of the arguments to `Module:handle_event/2`.

`notify` will not fail even if the specified event manager does not exist, unless it is specified as `Name`.

```
call(EventMgrRef, Handler, Request) -> Result
call(EventMgrRef, Handler, Request, Timeout) -> Result
```

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Request = term()
- Timeout = int()>0 | infinity
- Result = Reply | {error,Error}
- Reply = term()
- Error = bad_module | {'EXIT',Reason} | term()
- Reason = term()

Makes a synchronous call to the event handler `Handler` installed in the event manager `EventMgrRef` by sending a request and waiting until a reply arrives or a timeout occurs. The event manager will call `Module:handle_call/2` to handle the request.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_call/2`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value `Reply` is defined in the return value of `Module:handle_call/2`. If the specified event handler is not installed, the function returns `{error,bad_module}`. If the callback function fails with `Reason` or returns an unexpected value `Term`, this function returns `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively.

```
delete_handler(EventMgrRef, Handler, Args) -> Result
```

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = term() | {error,module_not_found} | {'EXIT',Reason}
- Reason = term()

Deletes an event handler from the event manager `EventMgrRef`. The event manager will call `Module:terminate/2` to terminate the event handler.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`Args` is an arbitrary term which is passed as one of the arguments to `Module:terminate/2`.

The return value is the return value of `Module:terminate/2`. If the specified event handler is not installed, the function returns `{error,module_not_found}`. If the callback function fails with `Reason`, the function returns `{'EXIT',Reason}`.

`swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()
- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
- Reason = term()

Replaces an old event handler with a new event handler in the event manager `EventMgrRef`.

See `add_handler/3` for a description of the arguments.

First the old event handler `Handler1` is deleted. The event manager calls `Module1:terminate(Args1, ...)`, where `Module1` is the callback module of `Handler1`, and collects the return value.

Then the new event handler `Handler2` is added and initiated by calling `Module2:init({Args2,Term})`, where `Module2` is the callback module of `Handler2` and `Term` the return value of `Module1:terminate/2`. This makes it possible to transfer information from `Handler1` to `Handler2`.

The new handler will be added even if the the specified old event handler is not installed in which case `Term=error`, or if `Module1:terminate/2` fails with `Reason` in which case `Term={'EXIT',Reason}`. The old handler will be deleted even if `Module2:init/1` fails.

If there was a supervised connection between `Handler1` and a process `Pid`, there will be a supervised connection between `Handler2` and `Pid` instead.

If `Module2:init/1` returns a correct value, this function returns `ok`. If `Module2:init/1` fails with `Reason` or returns an unexpected value `Term`, this this function returns `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively.

`swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler 2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()
- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
- Reason = term()

Replaces an event handler in the event manager `EventMgrRef` in the same way as `swap_handler/3` but will also supervise the connection between `Handler2` and the calling process.

See `swap_handler/3` for a description of the arguments and return values.

`which_handlers(EventMgrRef) -> [Handler]`

Types:

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

Returns a list of all event handlers installed in the event manager `EventMgrRef`.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`stop(EventMgrRef) -> ok`

Types:

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

Terminates the event manager `EventMgrRef`. Before terminating, the event manager will call `Module:terminate(stop,...)` for each installed event handler.

See `add_handler/3` for a description of the argument.

CALLBACK FUNCTIONS

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

Exports

`Module:init(InitArgs) -> {ok,State}`

Types:

- InitArgs = Args | {Args,Term}
- Args = Term = term()
- State = term()

Whenever a new event handler is added to an event manager, this function is called to initialize the event handler.

If the event handler is added due to a call to `gen_event:add_handler/3` or `gen_event:add_sup_handler/3`, `InitArgs` is the `Args` argument of these functions.

If the event handler is replacing another event handler due to a call to `gen_event:swap_handler/3` or `gen_event:swap_sup_handler/3`, or due to a `swap` return tuple from one of the other callback functions, `InitArgs` is a tuple `{Args,Term}` where `Args` is the argument provided in the function call/return tuple and `Term` is the result of terminating the old event handler, see `gen_event:swap_handler/3`.

The function should return `{ok,State}` where `State` is the initial internal state of the event handler.

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

Types:

- Event = term()
- State = term()
- Result = {ok,NewState}
- | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
- NewState = term()
- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

Whenever an event manager receives an event sent using `gen_event:notify/2` or `gen_event:sync_notify/2`, this function is called for each installed event handler to handle the event.

`Event` is the `Event` argument of `notify/sync_notify`.

`State` is the internal state of the event handler.

If the function returns `{ok,NewState}` the event handler will remain in the event manager with the possible updated internal state `NewState`.

If the function returns `{swap_handler,Args1,NewState,Handler2,Args2}` the event handler will be replaced by `Handler2` by first calling `Module:terminate(Args1,NewState)` and then `Module2:init({Args2,Term})` where `Term` is the return value of `Module:terminate/2`. See `gen_event:swap_handler/3` for more information.

If the function returns `remove_handler` the event handler will be deleted by calling `Module:terminate(remove_handler,State)`.

`Module:handle_call(Request, State) -> Result`

Types:

- Request = term()
- State = term()
- Result = {ok,Reply,NewState}
- | {swap_handler,Reply,Args1,NewState,Handler2,Args2}
- | {remove_handler, Reply}
- Reply = term()
- NewState = term()
- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

Whenever an event manager receives a request sent using `gen_event:call/3,4`, this function is called for the specified event handler to handle the request.

`Request` is the `Request` argument of `call`.

`State` is the internal state of the event handler.

The return values are the same as for `handle_event/2` except they also contain a term `Reply` which is the reply given back to the client as the return value of `call`.

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

Types:

- Info = term()
- State = term()
- Result = {ok,NewState}
- | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
- NewState = term()
- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

This function is called for each installed event handler when an event manager receives any other message than an event or a synchronous request (or a system message).

`Info` is the received message.

See `Module:handle_event/2` for a description of State and possible return values.

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

Types:

- Arg = Args | {stop,Reason} | stop | remove_handler
- | {error,{'EXIT',Reason}} | {error,Term}
- Args = Reason = Term = term()

Whenever an event handler is deleted from an event manager, this function is called. It should be the opposite of `Module:init/1` and do any necessary cleaning up.

If the event handler is deleted due to a call to `gen_event:delete_handler`, `gen_event:swap_handler/3` or `gen_event:swap_sup_handler/3`, `Arg` is the `Args` argument of this function call.

`Arg={stop,Reason}` if the event handler has a supervised connection to a process which has terminated with reason `Reason`.

`Arg=stop` if the event handler is deleted because the event manager is terminating.

`Arg=remove_handler` if the event handler is deleted because another callback function has returned `remove_handler` or `{remove_handler,Reply}`.

`Arg={error,Term}` if the event handler is deleted because a callback function returned an unexpected value `Term`, or `Arg={error,{'EXIT',Reason}}` if a callback function failed.

`State` is the internal state of the event handler.

The function may return any term. If the event handler is deleted due to a call to `gen_event:delete_handler`, the return value of that function will be the return value of this function. If the event handler is to be replaced with another event handler due to a swap, the return value will be passed to the `init` function of the new event handler. Otherwise the return value is ignored.

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

Types:

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

This function is called for each installed event handler they should update the internal state due to code replacement, i.e. when the instruction `{update,Module,Change,PrePurge,PostPurge,Modules` where `Change={advanced,Extra}` has been given to the release handler. See *SASL User's Guide* for more information.

`OldVsn` is the `vsn` attribute of the old version of the callback module `Module`, or `undefined` if no such attribute is defined.

`State` is the internal state of the event handler.

`Extra` is the same as in the `{advanced,Extra}` part of the update instruction.

The function should return `{ok,NewState}`, where `NewState` is the updated internal state.

SEE ALSO

supervisor(3) [page 290], sys(3) [page 300]

gen_fsm

Erlang Module

A behaviour module for implementing a finite state machine. A generic finite state machine process (gen_fsm) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to *OTP Design Principles* for more information.

A gen_fsm assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_fsm module                    Callback module
--------------                    ---------------
gen_fsm:start_link                -----> Module:init/1

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

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

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

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

-                                 -----> Module:handle_info/3

-                                 -----> Module:terminate/3

-                                 -----> Module:code_change/4
```

If a callback function fails or returns a bad value, the gen_fsm will terminate.

The sys module can be used for debugging a gen_fsm.

Note that a gen_fsm does not trap exit signals automatically, this must be explicitly initiated in the callback module.

Unless otherwise stated, all functions in this module fail if the specified gen_fsm does not exist or if bad arguments are given.

Exports

```
start(Module, Args, Options) -> Result
start(FsmName, Module, Args, Options) -> Result
start_link(Module, Args, Options) -> Result
start_link(FsmName, Module, Args, Options) -> Result
```

Types:

- FsmName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [term()]
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a gen_fsm process which calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, this function does not return until `Module:init/1` has returned.

A gen_fsm started using `start_link` is linked to the calling process, this function must be used if the gen_fsm is included in a supervision tree. A gen_fsm started using `start` is not linked to the calling process.

If `FsmName={local,Name}`, the gen_fsm is registered locally as `Name` using `register/2`. If `FsmName={global,Name}`, the gen_fsm is registered globally as `Name` using `global:register_name/2`. If no name is provided, the gen_fsm is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_fsm is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding `sys` function will be called for each item in `Dbgs`. Refer to `sys(3)` for more information.

If the option `{spawn_opt,SOpts}` is present, `SOpts` will be passed as option list to the `spawn_opt` BIF which is used to spawn the gen_fsm process. Refer to `erlang(3)` for information about the `spawn_opt` options.

If the gen_fsm is successfully created and initialized the function returns `{ok,Pid}`, where `Pid` is the pid of the gen_fsm. If there already exists a process with the specified `FsmName`, the function returns `{error,{already_started,Pid}}` where `Pid` is the pid of that process.

If `Module:init/1` fails with `Reason`, the function returns `{error,Reason}`. If `Module:init/1` returns `{stop,Reason}` or `ignore`, the process is terminated and the function returns `{error,Reason}` or `ignore`, respectively.

`send_event(FsmRef, Event) -> ok`

Types:

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

Sends an event asynchronously to the gen_fsm `FsmRef` and returns `ok` immediately. The gen_fsm will call `Module:StateName/2` to handle the event, where `StateName` is the name of the current state of the gen_fsm.

`FsmRef` can be:

- the pid,
- `Name`, if the gen_fsm is locally registered,
- {`Name,Node`}, if the gen_fsm is locally registered at another node, or
- {`global,Name`}, if the gen_fsm is globally registered.

`Event` is an arbitrary term which is passed as one of the arguments to `Module:StateName/2`.

`send_all_state_event(FsmRef, Event) -> ok`

Types:

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

Sends an event asynchronously to the gen_fsm `FsmRef` and returns `ok` immediately. The gen_fsm will call `Module:handle_event/3` to handle the event.

See `send_event/2` for a description of the arguments.

The difference between `send_event` and `send_all_state_event` is which callback function is used to handle the event. This function is useful when sending events that are handled the same way in every state, as only one `handle_event` clause is needed to handle the event instead of one clause in each state name function.

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

Types:

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

Sends an event to the gen_fsm `FsmRef` and waits until a reply arrives or a timeout occurs. The gen_fsm will call `Module:StateName/3` to handle the event, where `StateName` is the name of the current state of the gen_fsm.

See `send_event/2` for a description of `FsmRef` and `Event`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value `Reply` is defined in the return value of `Module:StateName/3`.

In the case where the gen_fsm terminates during the handling of the event and the caller is linked to the gen_fsm and trapping exits, the exit message is removed from the caller's receive queue before the function call fails.
This behaviour is retained for backwards compatibility only and may change in the future. Note that if the gen_fsm crashes in between calls, a linked process must take care of the exit message anyway.
Warning: Under certain circumstances (e.g. `FsmRef = {Name,Node}`, and `Node` goes down) the exit message cannot be removed.

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

Types:

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

Sends an event to the gen_fsm `FsmRef` and waits until a reply arrives or a timeout occurs. The gen_fsm will call `Module:handle_sync_event/4` to handle the event.

See `send_event/2` for a description of `FsmRef` and `Event`. See `sync_send_event/3` for a description of `Timeout` and `Reply`.

See `send_all_state_event/2` for a discussion about the difference between `sync_send_event` and `sync_send_all_state_event`.

`reply(Caller, Reply) -> true`

Types:

- Caller - see below
- Reply = term()

This function can be used by a gen_fsm to explicitly send a reply to a client process that called `sync_send_event` or `sync_send_all_state_event`, when the reply cannot be defined in the return value of `Module:State/3` or `Module:handle_sync_event/4`.

`Caller` must be the `From` argument provided to the callback function. `Reply` is an arbitrary term, which will be given back to the client as the return value of `sync_send_event` or `sync_send_all_state_event`.

`send_event_after(Time, Event) -> Ref`

Types:

- Time = integer()

- Event = term()
- Ref = reference()

Sends a delayed event internally in the gen_fsm that calls this function after `Time` ms. Returns immediately a reference that can be used to cancel the delayed send using `cancel_timer/1`.

The gen_fsm will call `Module:StateName/2` to handle the event, where `StateName` is the name of the current state of the gen_fsm at the time the delayed event is delivered.

`Event` is an arbitrary term which is passed as one of the arguments to `Module:StateName/2`.

`start_timer(Time, Msg) -> Ref`

Types:

- Time = integer()
- Msg = term()
- Ref = reference()

Sends a timeout event internally in the gen_fsm that calls this function after `Time` ms. Returns immediately a reference that can be used to cancel the timer using `cancel_timer/1`.

The gen_fsm will call `Module:StateName/2` to handle the event, where `StateName` is the name of the current state of the gen_fsm at the time the timeout message is delivered.

`Msg` is an arbitrary term which is passed in the timeout message, `{timeout, Ref, Msg}`, as one of the arguments to `Module:StateName/2`.

`cancel_timer(Ref) -> RemainingTime | false`

Types:

- Ref = reference()
- RemainingTime = integer()

Cancels an internal timer referred by `Ref` in the gen_fsm that calls this function.

`Ref` is a reference returned from `send_event_after/2` or `start_timer/2`.

If the timer has already timed out, but the event not yet been delivered, it is cancelled as if it had *not* timed out, so there will be no false timer event after returning from this function.

Returns the remaining time in ms until the timer would have expired if `Ref` referred to an active timer, `false` otherwise.

CALLBACK FUNCTIONS

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

In the description, the expression *state name* is used to denote a state of the state machine. *state data* is used to denote the internal state of the Erlang process which implements the state machine.

Exports

`Module:init(Args) -> Result`

Types:

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

Whenever a gen_fsm is started using `gen_fsm:start/3,4` or `gen_fsm:start_link/3,4`, this function is called by the new process to initialize.

`Args` is the `Args` argument provided to the start function.

If initialization is successful, the function should return `{ok,StateName,StateData}` or `{ok,StateName,StateData,Timout}`, where `StateName` is the initial state name and `StateData` the initial state data of the gen_fsm.

If an integer timout value is provided, a timout will occur unless an event or a message is received within `Timeout` milliseconds. A timout is represented by the atom `timeout` and should be handled by the `Module:StateName/2` callback functions. The atom `inifinity` can be used to wait indefinitely, this is the default value.

If something goes wrong during the initialization the function should return `{stop,Reason}`, where `Reason` is any term, or `ignore`.

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

Types:

- Event = timeout | term()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout}
- | {stop,Reason,NewStateData}
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using `gen_fsm:send_event/2`, the instance of this function with the same name as the current state name `StateName` is called to handle the event. It is also called if a timeout occurs.

`Event` is either the atom `timeout`, if a timeout has occured, or the `Event` argument provided to `send_event`.

`StateData` is the state data of the gen_fsm.

If the function returns `{next_state,NextStateName,NewStateData}` or `{next_state,NextStateName,NewStateData,Timeout}`, the gen_fsm will continue executing with the current state name set to `NextStateName` and with the possibly updated state data `NewStateData`. See `Module:init/1` for a description of `Timeout`.

If the function returns `{stop,Reason,NewStateData}`, the gen_fsm will call `Module:terminate(Reason,NewStateData)` and terminate.

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

Types:

- Event = term()
- StateName = atom()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout}
- | {stop,Reason,NewStateData}
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

Whenever a gen_fsm receives an event sent using `gen_fsm:send_all_state_event/2`, this function is called to handle the event.

`StateName` is the current state name of the gen_fsm.

See `Module:StateName/2` for a description of the other arguments and possible return values.

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

Types:

- Event = term()
- From = {pid(),Tag}
- StateData = term()
- Result = {reply,Reply,NextStateName,NewStateData} | {reply,Reply,NextStateName,NewStateData,Timeout}
- | {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout}
- | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- Reply = term()
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = normal | term()

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using `gen_fsm:sync_send_event/2,3`, the instance of this function with the same name as the current state name `StateName` is called to handle the event.

`Event` is the `Event` argument provided to `sync_send_event`.

`From` is a tuple `{Pid,Tag}` where `Pid` is the pid of the process which called `sync_send_event` and `Tag` is a unique tag.

StateData is the state data of the gen_fsm.

If the function returns {reply,Reply,NextStateName,NewStateData} or {reply,Reply,NextStateName,NewStateData,Timeout}, Reply will be given back to From as the return value of sync_send_event. The gen_fsm then continues executing with the current state name set to NextStateName and with the possibly updated state data NewStateData. See Module:init/1 for a description of Timeout.

If the function returns {next_state,NextStateName,NewStateData} or {next_state,NextStateName,NewStateData,Timeout}, the gen_fsm will continue executing in NextStateName with NewStateData. Any reply to From must be given explicitly using gen_fsm:reply/2.

If the function returns {stop,Reason,Reply,NewStateData}, Reply will be given back to From. If the function returns {stop,Reason,NewStateData}, any reply to From must be given explicitly using gen_fsm:reply/2. The gen_fsm will then call Module:terminate(Reason,NewStateData) and terminate.

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

Types:

- Event = term()
- From = {pid(),Tag}
- StateName = atom()
- StateData = term()
- Result = {reply,Reply,NextStateName,NewStateData} | {reply,Reply,NextStateName,NewStateData,Timeout}
- | {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout}
- | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- Reply = term()
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_all_state_event/2,3, this function is called to handle the event.

StateName is the current state name of the gen_fsm.

See Module:StateName/3 for a description of the other arguments and possible return values.

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

Types:

- Info = term()
- StateName = atom()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout}
- | {stop,Reason,NewStateData}
- NextStateName = atom()

- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = normal | term()

This function is called by a gen_fsm when it receives any other message than a synchronous or asynchronous event (or a system message).

`Info` is the received message.

See `Module:StateName/2` for a description of the other arguments and possible return values.

`Module:terminate(Reason, StateName, StateData)`

Types:

- Reason = normal | shutdown | term()
- StateName = atom()
- StateData = term()

This function is called by a gen_fsm when it is about to terminate. It should be the opposite of `Module:init/1` and do any necessary cleaning up. When it returns, the gen_fsm terminates with `Reason`. The return value is ignored.

`Reason` is a term denoting the stop reason, `StateName` is the current state name, and `StateData` is the state data of the gen_fsm.

`Reason` depends on why the gen_fsm is terminating. If it is because another callback function has returned a stop tuple `{stop,..}`, `Reason` will have the value specified in that tuple. If it is due to a failure, `Reason` is the error reason.

If the gen_fsm is part of a supervision tree and is ordered by its superviser to terminate, this function will be called with `Reason=shutdown` if the following conditions apply:

- the gen_fsm has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor's child specification is an integer timeout value, not `brutal_kill`.

Otherwise, the gen_fsm will be immediately terminated.

Note that for any other reason than `normal` or `shutdown`, the gen_fsm is assumed to terminate due to an error and an error report is issued using `error_logger:format/2`.

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

Types:

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

This function is called by a gen_fsm when it should update its state data due to a code replacement, i.e. when the instruction {`update,Module,Change,PrePurge,PostPurge,Modules`} where `Change=`{`advanced,Extra`} has been given to the release handler. See *SASL User's Guide* for more information.

`OldVsn` is the `vsn` attribute of the old version of the callback module `Module`, or `undefined` if no such attribute is defined.

`StateName` is the current state name and `StateData` the state data of the gen_fsm.

`Extra` is the same as in the {`advanced,Extra`} part of the update instruction.

The function should return the new current state name and updated state data.

SEE ALSO

supervisor(3) [page 290], sys(3) [page 300]

gen_server

Erlang Module

A behaviour module for implementing the server of a client-server relation. A generic server process (gen_server) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to *OTP Design Principles* for more information.

A gen_server assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_server module            Callback module
-----------------            ---------------
gen_server:start      -----> Module:init/1

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

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

-                     -----> Module:handle_info/2

-                     -----> Module:terminate/2

-                     -----> Module:code_change/3
```

If a callback function fails or returns a bad value, the gen_server will terminate.

The `sys` module can be used for debugging a gen_server.

Note that a gen_server does not trap exit signals automatically, this must be explicitly initiated in the callback module.

Unless otherwise stated, all functions in this module fail if the specified gen_server does not exist or if bad arguments are given.

Exports

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

Types:

- ServerName = {local,Name} | {global,Name}

- Name = atom()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [term()]
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a gen_server process which calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, this function does not return until `Module:init/1` has returned.

A gen_server started using `start_link` is linked to the calling process, this function must be used if the gen_server is included in a supervision tree. A gen_server started using `start` is not linked to the calling process.

If `ServerName={local,Name}` the gen_server is registered locally as `Name` using `register/2`. If `ServerName={global,Name}` the gen_server is registered globally as `Name` using `global:register_name/2`. If no name is provided, the gen_server is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_server is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding `sys` function will be called for each item in `Dbgs`. Refer to `sys(3)` for more information.

If the option `{spawn_opt,SOpts}` is present, `SOpts` will be passed as option list to the `spawn_opt` BIF which is used to spawn the gen_server. Refer to `erlang(3)` for information about the `spawn_opt` options.

If the gen_server is successfully created and initialized the function returns `{ok,Pid}`, where `Pid` is the pid of the gen_server. If there already exists a process with the specified `ServerName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

If `Module:init/1` fails with `Reason`, the function returns `{error,Reason}`. If `Module:init/1` returns `{stop,Reason}` or `ignore`, the process is terminated and the function returns `{error,Reason}` or `ignore`, respectively.

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

Types:

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

- Timeout = int()>0 | infinity
- Reply = term()

Makes a synchronous call to the gen_server `ServerRef` by sending a request and waiting until a reply arrives or a timout occurs. The gen_server will call `Module:handle_call/3` to handle the request.

`ServerRef` can be:

- the pid,
- `Name`, if the gen_server is locally registered,
- {`Name,Node`}, if the gen_server is locally registered at another node, or
- {`global,GlobalName`}, if the gen_server is globally registered.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_call/3`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value `Reply` is defined in the return value of `Module:handle_call/3`.

The call may fail for several reasons, including timeout and the called gen_server dying before or during the call.

There is a special case for backwards compatibility. If

- the client is linked to the gen server, and
- the client is trapping exits, and
- the gen_server terminates while handling the request

then the exit message is removed from the client's receive queue before the function call fails. This special-case behaviour may be removed in the future because it is inconsistent with the behaviour when a gen_server dies between calls and also because the exit message cannot be removed in some circumstances, for instance when `ServerRef = {Name, Node}` and `Node` goes down.

```
multi_call(Name, Request) -> Result
multi_call(Nodes, Name, Request) -> Result
multi_call(Nodes, Name, Request, Timeout) -> Result
```

Types:

- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()
- Timeout = int()>=0 | infinity
- Result = {Replies,BadNodes}
- Replies = [{Node,Reply}]
- Reply = term()
- BadNodes = [Node]

Makes a synchronous call to all gen_servers locally registered as `Name` at the specified nodes by first sending a request to every node and then waiting for the replies. The gen_servers will call `Module:handle_call/3` to handle the request.

The function returns a tuple `{Replies,BadNodes}` where `Replies` is a list of `{Node,Reply}` and `BadNodes` is a list of node that either did not exist, or where the gen_server `Name` did not exist or did not reply.

`Nodes` is a list of node names to which the request should be sent. Default value is the list of all known nodes `[node()|nodes()]`.

`Name` is the locally registered name of each gen_server.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_call/3`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for each reply, or the atom `infinity` to wait indefinitely. Default value is `infinity`. If no reply is received from a node within the specified time, the node is added to `BadNodes`.

When a reply `Reply` is received from the gen_server at a node `Node`, `{Node,Reply}` is added to `Replies`. `Reply` is defined in the return value of `Module:handle_call/3`.

> **Warning:**
> If one of the nodes is running Erlang/OTP R6B or older, and the gen_server is not started when the requests are sent, but starts within 2 seconds, this function waits the whole `Timeout`, which may be infinity.
>
> This problem does not exist if all nodes are running Erlang/OTP R7B or later.

This function does *not* read out any exit messages like `call/2,3` does.

The previously undocumented functions `safe_multi_call/2,3,4` were removed in OTP R7B/Erlang 5.0 since this function is now safe, except in the case mentioned above.

To avoid that late answers (after the timeout) pollutes the caller's message queue, a middleman process is used to do the actual calls. Late answers will then be discarded when they arrive to a terminated process.

`cast(ServerRef, Request) -> ok`

Types:

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

Sends an asynchronous request to the gen_server `ServerRef` and returns `ok` immediately, ignoring if the destination node or gen_server does not exist. The gen_server will call `Module:handle_cast/2` to handle the request.

See `call/2,3` for a description of `ServerRef`.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_cast/2`.

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

Types:

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

Sends an asynchronous request to the gen_servers locally registered as `Name` at the specified nodes. The function returns immediately and ignores nodes that does not exist, or where the gen_server `Name` does not exist. The gen_servers will call `Module:handle_cast/2` to handle the request.

See `multi_call/2,3,4` for a description of the arguments.

```
reply(Client, Reply) -> true
```

Types:

- Client - see below
- Reply = term()

This function can be used by a gen_server to explicitly send a reply to a client that called `call` or `multi_call`, when the reply cannot be defined in the return value of `Module:handle_call/3`.

`Client` must be the `From` argument provided to the callback function. `Reply` is an arbitrary term, which will be given back to the client as the return value of `call` or `multi_call`.

```
enter_loop(Module, Options, State, ServerName, Timeout) ->
enter_loop(Module, Options, State, Timeout) ->
enter_loop(Module, Options, State, ServerName) -> _
```

Types:

- ServerName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Options = [Option]
- Option = {debug,Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics | {log_to_file,FileName} | {install,{Func,FuncState}}
- State = term()
- Timeout = int()>0 | infinity

Makes an existing process into a gen_server. The calling process will enter the gen_server receive loop and become a gen_server process. The process *must* have been started using one of the start functions in proc_lib, see proc_lib(3). The user is responsible for any initialization of the process, including registering a name for it.

This function is useful when a more complex initalization procedure is needed than the gen_server behaviour provides.

ServerName, Module and Options have the same meanings as when calling gen_server:start[_link]/3,4. However, if a `ServerName` is specified, the process must have been registered accordingly *before* this function is called, otherwise the process is terminated.

State and Timeout have the same meanings as in the return value of Module:init/1. Also, the callback module Module does not need to export an init/1 function.

CALLBACK FUNCTIONS

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

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,State} | {ok,State,Timeout}
- | {stop,Reason} | ignore
- State = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server is started using `gen_server:start/3,4` or `gen_server:start_link/3.4`, this function is called by the new process to initialize.

`Args` is the `Args` argument provided to the start function.

If the initialization is successful, the function should return `{ok,State}` or `{ok,State,Timout}`, where `State` is the internal state of the gen_server.

If an integer timout value is provided, a timout will occur unless a request or a message is received within `Timeout` milliseconds. A timout is represented by the atom `timeout` which should be handled by the `handle_info/2` callback function. The atom `inifinity` can be used to wait indefinitely, this is the default value.

If something goes wrong during the initialization the function should return `{stop,Reason}` where `Reason` is any term, or `ignore`.

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

Types:

- Request = term()
- From = {pid(),Tag}
- State = term()

- Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout}
- | {noreply,NewState} | {noreply,NewState,Timeout}
- | {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
- Reply = term()
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server receives a request sent using `gen_server:call/2,3` or `gen_server:multi_call/2,3,4`, this function is called to handle the request.

`Request` is the `Request` argument provided to `call` or `multi_call`.

`From` is a tuple {`Pid,Tag`} where `Pid` is the pid of the client and `Tag` is a unique tag.

`State` is the internal state of the gen_server.

If the function returns {`reply,Reply,NewState`} or {`reply,Reply,NewState,Timout`}, `Reply` will be given back to `From` as the return value of `call` or included in the return value of `multi_call`. The gen_server then continues executing with the possibly updated internal state `NewState`. See `Module:init/1` for a description of `Timeout`.

If the functions returns {`noreply,NewState`} or {`noreply,NewState,Timeout`}, the gen_server will continue executing with `NewState`. Any reply to `From` must be given explicitly using `gen_server:reply/2`.

If the function returns {`stop,Reason,Reply,NewState`}, `Reply` will be given back to `From`. If the function returns {`stop,Reason,NewState`}, any reply to `From` must be given explicitly using `gen_server:reply/2`. The gen_server will then call `Module:terminate(Reason,NewState)` and terminate.

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

Types:

- Request = term()
- State = term()
- Result = {noreply,NewState} | {noreply,NewState,Timeout}
- | {stop,Reason,NewState}
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server receives a request sent using `gen_server:cast/2` or `gen_server:abcast/2,3`, this function is called to handle the request.

See `Module:handle_call/3` for a description of the arguments and possible return values.

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

Types:

- Info = timeout | term()
- State = term()
- Result = {noreply,NewState} | {noreply,NewState,Timeout}
- | {stop,Reason,NewState}

- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = normal | term()

This function is called by a gen_server when a timeout occurs or when it receives any other message than a synchronous or asynchronous request (or a system message).

`Info` is either the atom `timeout`, if a timeout has occured, or the received message.

See `Module:handle_call/3` for a description of the other arguments and possible return values.

`Module:terminate(Reason, State)`

Types:

- Reason = normal | shutdown | term()
- State = term()

This function is called by a gen_server when it is about to terminate. It should be the opposite of `Module:init/1` and do any necessary cleaning up. When it returns, the gen_server terminates with `Reason`. The return value is ignored.

`Reason` is a term denoting the stop reason and `State` is the internal state of the gen_server.

`Reason` depends on why the gen_server is terminating. If it is because another callback function has returned a stop tuple `{stop,..}`, `Reason` will have the value specified in that tuple. If it is due to a failure, `Reason` is the error reason.

If the gen_server is part of a supervision tree and is ordered by its supervisor to terminate, this function will be called with `Reason=shutdown` if the following conditions apply:

- the gen_server has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor's child specification is an integer timeout value, not `brutal_kill`.

Otherwise, the gen_server will be immediately terminated.

Note that for any other reason than `normal` or `shutdown`, the gen_server is assumed to terminate due to an error and an error report is issued using `error_logger:format/2`.

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

Types:

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

This function is called by a gen_server when it should update its internal state due to code replacement, i.e. when the instruction {`update,Module,Change,PrePurge,PostPurge,Modules`} where `Change=`{`advanced,Extra`} has been given to the release handler. See *SASL User's Guide* for more information.

`OldVsn` is the `vsn` attribute of the old version of the callback module `Module`, or `undefined` if no such attribute is defined.

`State` is the internal state of the gen_server.

`Extra` is the same as in the {`advanced,Extra`} part of the update instruction.

The function should return the updated internal state.

SEE ALSO

supervisor(3) [page 290], sys(3) [page 300]

io

Erlang 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, unfortunately now very outdated, but the general principles still apply.

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.

`setopts([IoDevice,] OptList)`

Set options for standard input/output (`IoDevice`). Possible options are:

`binary` Makes `get_chars/2,3` and `get_line/1,2` return binaries instead of lists of chars.

`list` Makes `get_chars/2,3` and `get_line/1,2` return lists of chars, which is the default.

Returns `ok` if succesful or {`error,Reason`} if not.

> **Note:**
> The `binary` option does not work against I/O servers on remote nodes running an older version of Erlang/OTP than R9C.

`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
```

B Writes an integer in base 2..36, the default base is 10. A leading dash is printed for negative integers.

The precision field selects base. For example:

```
>io:format("~.16B~n", [31]).
1F
ok
>io:format("~.2B~n", [-19]).
-10011
ok
>io:format("~.36B~n", [5*36+35]).
5Z
ok
```

X Like B, but takes an extra argument that is a prefix to insert before the number, but after the leading dash, if any.

The prefix can be a possibly deep list of character codes or an atom.

```
>io:format("~X~n", [31,"10#"]).
10#31
ok
>io:format("~.16X~n", [-31,"0x"]).
-0x1F
ok
```

Like B, but prints the number with an Erlang style '#'-separated base prefix.

```
>io:format("~.10#~n", [31]).
10#31
ok
>io:format("~.16X~n", [-31]).
-16#1F
ok
```

b Like B, but prints lowercase letters.

x Like X, but prints lowercase letters.

+ Like #, but prints lowercase letters.

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,[<0.20.0>,"~s","A"]},
                    {erl_eval,expr,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.

  `u` An unsigned integer in base 2..36 is expected. The field width parameter is used to specify base. Leading white-space characters are not skipped.

  `-` An optional sign character is expected. A sign character '-' gives the return value `-1`. Sign character '+' or none gives `1`. The field width parameter is ignored. Leading white-space characters are not skipped.

  `#` An integer in base 2..36 with Erlang-style base prefix (for example `"16#ffff"`) 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

Erlang 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(3) [page 182] 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(3) [page 182] 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

Erlang 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

Erlang 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]
```

The result need not be a proper list. The last parameter may be of any datatype and will be the tail in the resulting list. An example:

```
> lists:append([[a,b],c]).
[a,b|c]
```

The atom `c` will be the tail of the list and the list is therefore not proper (a proper list ends with `[]`).

As a parameter of `[]` is ignored this example is also valid (although probably useless):

```
lists:append([[],d]).
```

`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`.

The behaviour regarding inproper lists is identical to the behaviour of `lists:append/1` `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 `Nth` 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 `Nth` element is `Key`.

`keymerge(N, List1, List2)`

Types:

- N = int()
- List1 = List2 = [tuple()]

Returns the sorted list formed by merging `List1` and `List2`. The merge is performed on the `Nth` element of each tuple. Both `List1` and `List2` must be key-sorted prior to evaluating this function; otherwise the result is undefined (and probably unexpected). 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 `Nth` 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(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns the sorted list formed by merging all the sub-lists of `ListOfLists`. All sub-lists must be sorted prior to evaluating this function.

`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 according to the ordering function `Fun` prior to evaluating this function. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`merge3(List1, List2, List3) -> List4`

Types:

- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging `List1`, `List2` and `List3`. All of `List1`, `List2` and `List3` must be sorted prior to evaluating this function.

`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`.

`ukeymerge(N, List1, List2)`

Types:

- N = int()
- List1 = List2 = [tuple()]

Returns the sorted list formed by merging `List1` and `List2` while removing consecutive duplicates. 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`.

`ukeysort(N, List1) -> List2`

Types:

- N = int()
- List1 = List2 = [tuple()]

Returns a list containing the sorted elements of `List1` with consecutive duplicates removed. `TupleList1` must be a list of tuples, and the sort is performed on the `N`th element of the tuple. The sort is stable.

`umerge(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns the sorted list formed by merging all the sub-lists of `ListOfLists` while removing duplicates. All sub-lists must be sorted and contain no duplicates prior to evaluating this function.

`umerge(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging `List1` and `List2` while removing duplicates. Both `List1` and `List2` must be sorted and contain no duplicates prior to evaluating this function.

`umerge(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` while removing consecutive duplicates. Both `List1` and `List2` must be sorted according to the ordering function `Fun` prior to evaluating this function. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`umerge3(List1, List2, List3) -> List4`

Types:

- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging `List1`, `List2` and `List3` while removing duplicates. All of `List1`, `List2` and `List3` must be sorted and contain no duplicates prior to evaluating this function.

`usort(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list which contains the sorted elements of `List1` without duplicates.

`usort(Fun, List1) -> List2`

Types:

- List1 = List2 = [Element]
- Fun = fun(Element, Element) -> bool()
- Element = term()

Returns a list which contains the sorted elements of `List1` with consecutive duplicates removed, according to the ordering function `Fun`. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`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 `A`s to `B`s, and a list of `A`s and produces a list of `B`s 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.

`split(N, List) -> {List1, List2}`

Types:

- Pred = fun(A) -> bool()
- List = List1 = List2 = [A]

Partitions `List` into `List1` and `List2`. `List1` contains the first `N` elements and `List2` the rest of the elements.

Note that `List == List1 ++ List2`.

`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

Erlang 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) [page 154], rb(3)

math

Erlang 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(Y, X)
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.

ms_transform

Erlang Module

This module implements the parse_transform that makes calls to ets and dbg:fun2ms/1 translate into literal match specifications. It also implements the back end for the same functions when called from the Erlang shell.

The translations from fun's to match_specs is accessed through the two "pseudo functions" ets:fun2ms/1 and dbg:fun2ms/1.

Actually this introduction is more or less an introduction to the whole concept of match specifications. Since everyone trying to use ets:select or dbg seems to end up reading this page, it seems in good place to explain a little more than just what this module does.

There are some caveats one should be aware of, please read through the whole manual page if it's the first time you're using the transformations.

Match specifications are used more or less as filters. They resemble usual Erlang matching in a list comprehension or in a fun used in conjunction with lists:foldl etc. The syntax of pure match specifications is somewhat awkward though, as they are made up purely by Erlang terms and there is no syntax in the language to make the match specifications more readable.

As the match specifications execution and structure is quite like that of a fun, it would for most programmers be more straight forward to simply write it using the familiar fun syntax and having that translated into a match specification automatically. Of course a real fun is more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it's still more convenient to write it all as a fun. This module contains the code that simply translates the fun syntax into match_spec terms.

Let's start with an ets example. Using ets:select and a match specification, one can filter out rows of a table and construct a list of tuples containing relevant parts of the data in these rows. Of course one could use ets:foldl instead, but the select call is far more efficient. Without the translation, one has to struggle with writing match specifications terms to accommodate this, or one has to resort to the less powerful ets:match(_object) calls, or simply give up and use the more inefficient method of ets:foldl. Using the ets:fun2ms transformation, a ets:select call is at least as easy to write as any of the alternatives.

As an example, consider a simple table of employees:

```
-record(emp, {empno,     %Employee number as a string, the key
              surname,   %Surname of the employee
              givenname, %Given name of employee
              dept,      %Department one of {dev,sales,prod,adm}
              empyear}). %Year the employee was employed
```

We create the table using:

```
ets:new(emp_tab,[{keypos,#emp.empno},named_table,ordered_set]).
```

Let's also fill it with some randomly chosen data for the examples:

```
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
 {emp,"122334","Weston","Anna",prod,2002},
 {emp,"535216","Chalker","Samuel",adm,1998},
 {emp,"789789","Harrysson","Joe",adm,1996},
 {emp,"963721","Scott","Juliana",dev,2003},
 {emp,"989891","Brown","Gabriel",prod,1999}]
```

Now, the amount of data in the table is of course to small to justify complicated ets searches, but on real tables, using `select` to get exactly the data you want will increase efficiency remarkably.

Lets say for example that we'd want the employee numbers of everyone in the sales department. One might use `ets:match` in such a situation:

```
1> ets:match(emp_tab, {'_', '$1', '_', '_', sales, '_'}).
[["011103"],["076324"]]
```

Even though `ets:match` does not require a full match specification, but a simpler type, it's still somewhat unreadable, and one has little control over the returned result, it's always a list of lists. OK, one might use `ets:foldl` or `ets:foldr` instead:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
             (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

Running that would result in `["011103","076324"]` , which at least gets rid of the extra lists. The fun is also quite straightforward, so the only problem is that all the data from the table has to be transferred from the table to the calling process for filtering. That's inefficient compared to the `ets:match` call where the filtering can be done “inside” the emulator and only the result is transferred to the process. Remember that ets tables are all about efficiency, if it wasn't for efficiency all of ets could be implemented in Erlang, as a process receiving requests and sending answers back. One uses ets because one wants performance, and therefore one wouldn't want all of the table transferred to the process for filtering. OK, let's look at a pure `ets:select` call that does what the `ets:foldr` does:

```
ets:select(emp_tab,[{#emp{empno = '$1', dept = sales, _='_'},[],['$1']}]).
```

Even though the record syntax is used, it's still somewhat hard to read and even harder to write. The first element of the tuple, `#emp{empno = '$1', dept = sales, _='_'}` tells what to match, elements not matching this will not be returned at all, as in the `ets:match` example. The second element, the empty list is a list of guard expressions, which we need none, and the third element is the list of expressions constructing the return value (in ets this almost always is a list containing one single term). In our case `'$1'` is bound to the employee number in the head (first element of tuple), and hence it is the employee number that is returned. The result is `["011103","076324"]`, just as in the `ets:foldr` example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

We have one efficient but hardly readable way of doing it and one inefficient but fairly readable (at least to the skilled Erlang programmer) way of doing it. With the use of ets:fun2ms, one could have something that is as efficient as possible but still is written as a filter using the fun syntax:

```
-include_lib("stdlib/include/ms_transform.hrl").

% ...

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, dept = sales}) ->
                              E
                      end)).
```

This may not be the shortest of the expressions, but it requires no special knowledge of match specifications to read. The fun's head should simply match what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all data of the table to the process for filtering as in the `ets:foldr` example. In fact it's even easier to read then the `ets:foldr` example, as the select call in itself discards anything that doesn't match, while the fun of the `foldr` call needs to handle both the elements matching and the ones not matching.

It's worth noting in the above `ets:fun2ms` example that one needs to include `ms_transform.hrl` in the source code, as this is what triggers the parse transformation of the `ets:fun2ms` call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell of course) and therefore will take no resources at all in runtime. So although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

Let's look at some more `ets` examples. Let's say one wants to get all the employee numbers of any employee hired before the year 2000. Using `ets:match` isn't an alternative here as relational operators cannot be expressed there. Once again, an `ets:foldr` could do it (slowly, but correct):

```
ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
                    (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result will be `["052341","076324","535216","789789","989891"]`, as expected. Now the equivalent expression using a handwritten match specification would look something like this:

```
ets:select(emp_tab,[{#emp{empno = '$1', empyear = '$2', _='_'},
                     [{'<', '$2', 2000}],
                     ['$1']}]).
```

This gives the same result, the `[{'<', '$2', 2000}]` is in the guard part and therefore discards anything that does not have a empyear (bound to '$2' in the head) less than 2000, just as the guard in the `foldl` example. Lets jump on to writing it using `ets:fun2ms`

```
-include_lib("stdlib/include/ms_transform.hrl").

% ...

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, empyear = Y}) when Y < 2000 ->
                              E
                      end)).
```

Obviously readability is gained by using the parse transformation.

I'll show some more examples without the tiresome comparing-to-alternatives stuff. Let's say we'd want the whole object matching instead of only one element. We could of course assign a variable to every part of the record and build it up once again in the body of the `fun`, but it's easier to do like this:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(Obj = #emp{empno = E, empyear = Y})
                         when Y < 2000 ->
                              Obj
                      end)).
```

Just as in ordinary Erlang matching, you can bind a variable to the whole matched object using a “match in then match”, i.e. a =. Unfortunately this is not general in `fun's` translated to match specifications, only on the “top level”, i.e. matching the *whole* object arriving to be matched into a separate variable, is it allowed. For the one's used to writing match specifications by hand, I'll have to mention that the variable A will simply be translated into '$_'. It's not general, but it has very common usage, why it is handled as a special, but useful, case. If this bothers you, the pseudo function `object` also returns the whole matched object, see the part about caveats and limitations below.

Let's do something in the `fun`'s body too: Let's say that someone realizes that there are a few people having an employee number beginning with a zero (`0`), which shouldn't be allowed. All those should have their numbers changed to begin with a one (`1`) instead and one wants the list `[{<Old empno>,<New empno>}]` created:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = [$0 | Rest] }) ->
                              {[$0|Rest],[$1|Rest]}
                      end)).
```

As a matter of fact, this query hit's the feature of partially bound keys in the table type `ordered_set`, so that not the whole table need be searched, only the part of the table containing keys beginning with `0` is in fact looked into.

The fun of course can have several clauses, so that if one could do the following: For each employee, if he or she is hired prior to 1997, return the tuple `{inventory, <employee number>}`, for each hired 1997 or later, but before 2001, return `{rookie, <employee number>}`, for all others return `{newbie, <employee number>}`. All except for the ones named `Smith` as they would be affronted by anything other than the tag `guru` and that is also what's returned for their numbers; `{guru, <employee number>}`:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, surname = "Smith" }) ->
                              {guru,E};
                         (#emp{empno = E, empyear = Y}) when Y < 1997  ->
                              {inventory, E};
```

```
                    (#emp{empno = E, empyear = Y}) when Y > 2001  ->
                         {newbie, E};
                    (#emp{empno = E, empyear = Y}) -> % 1997 -- 2001
                         {rookie, E}
                end)).
```

The result will be:

```
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

and so the Smith's will be happy...

So, what more can you do? Well, the simple answer would be; look in the documentation of match specifications in ERTS users guide. However let's briefly go through the most useful "built in functions" that you can use when the `fun` is to be translated into a match specification by `ets:fun2ms` (it's worth mentioning, although it might be obvious to some, that calling other functions than the one's allowed in match specifications cannot be done. No "usual" Erlang code can be executed by the `fun` being translated by `fun2ms`, the `fun` is after all limited exactly to the power of the match specifications, which is unfortunate, but the price one has to pay for the execution speed of an `ets:select` compared to `ets:foldl/foldr`).

The head of the `fun` is obviously a head matching (or mismatching) *one* parameter, one object of the table we `select` from. The object is always a single variable (can be _) or a tuple, as that's what's in `ets, dets` and `mnesia` tables (the match specification returned by `ets:fun2ms` can of course be used with `dets:select` and `mnesia:select` as well as with `ets:select`). The use of = in the head is allowed (and encouraged) on the top level.

The guard section can contain any guard expression of Erlang, however one has to use the "new" names of type tests, the one's beginning with `is_`. As the new type tests (the `is_` tests) are in practice just guard bif's they can also be called from within the body of the fun, but so they can in ordinary Erlang code. Also arithmetics is allowed, as well as ordinary guard bif's. Here's a list of bif's and expressions:

- The type tests: is_atom, is_constant, is_float, is_integer, is_list, is_number, is_pid, is_port, is_reference, is_tuple, is_binary, is_function, is_record
- The boolean operators: not, and, or, andalso, orelse
- The relational operators: >, >=, <, =<, =:=, ==, =/=, /=
- Arithmetics: +, -, *, div, rem
- Bitwise operators: band, bor, bxor, bnot, bsl, bsr
- The guard bif's: abs, element, hd, length, node, round, size, tl, trunc, self

Contrary to the fact with “handwritten” match specifications, the is_record guard works as in ordinary Erlang code.

The body of the fun is used to construct the resulting value. When selecting from tables one usually just construct a suiting term here, using ordinary Erlang term construction, like tuple parentheses, list brackets and variables matched out in the head, possibly in conjunction with the occasional constant. Whatever expressions are allowed in guards are also allowed here, but there are no special functions except object and bindings (see further down), which returns the whole matched object and all known variable bindings respectively.

The dbg variants of match specifications have an imperative approach to the match specification body, the ets dialect hasn't. The fun body for ets:fun2ms returns the result without side effects, and as matching (=) in the body of the match specifications is not allowed (for performance reasons) the only thing left, more or less, is term construction...

Let's move on to the dbg dialect, the slightly different match specifications translated by dbg:fun2ms.

The same reasons for using the parse transformation applies to dbg, maybe even more so as filtering using Erlang code is simply not a good idea when tracing (except afterwards, if you trace to file). The concept is similar to that of ets:fun2ms except that you usually use it directly from the shell (which can also be done with ets:fun2ms).

Let's manufacture a toy module to trace on

```
-module(toy).

-export([start/1, store/2, retrieve/1]).

start(Args) ->
    toy_table = ets:new(toy_table,Args).

store(Key, Value) ->
    ets:insert(toy_table,{Key,Value}).

retrieve(Key) ->
    [{Key, Value}] = ets:lookup(toy_table,Key),
    Value.
```

During model testing, the first test bails out with a {badmatch,16} in {toy,start,1}, why?

We suspect the ets call, as we match hard on the return value, but want only the particular new call with toy_table as first parameter. So we start a default tracer on the node:

```
1> dbg:tracer().
{ok,<0.88.0>}
```

And so we turn on call tracing for all processes, we are going to make a pretty restrictive trace pattern, so there's no need to call trace only a few processes (it usually isn't):

```
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
```

It's time to specify the filter. We want to view calls that resemble ets:new(toy_table,<something>):

```
3> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

As can be seen, the `fun`'s used with `dbg:fun2ms` takes a single list as parameter instead of a single tuple. The list matches a list of the parameters to the traced function. A single variable may also be used of course. The body of the fun expresses in a more imperative way actions to be taken if the fun head (and the guards) matches. I return `true` here, but it's only because the body of a fun cannot be empty, the return value will be discarded.

When we run the test of our module now, we get the following trace output:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
```

Let's play we haven't spotted the problem yet, and want to see what `ets:new` returns. We do a slightly different trace pattern:

```
4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> return_trace() end)).
```

Resulting in the following trace output when we run the test:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
(<0.86.0>) returned from ets:new/2 -> 24
```

The call to `return_trace`, makes a trace message appear when the function returns. It applies only to the specific function call triggering the match specification (and matching the head/guards of the match specification). This is the by far the most common call in the body of a `dbg` match specification.

As the test now fails with `{badmatch,24}`, it's obvious that the badmatch is because the atom `toy_table` does not match the number returned for an unnamed table. So we spotted the problem, the table should be named and the arguments supplied by our test program does not include `named_table`. We rewrite the start function to:

```
start(Args) ->
    toy_table = ets:new(toy_table,[named_table |Args]).
```

And with the same tracing turned on, we get the following trace output:

```
(<0.86.0>) call ets:new(toy_table,[named_table,ordered_set])
(<0.86.0>) returned from ets:new/2 -> toy_table
```

Very well. Let's say the module now passes all testing and goes into the system. After a while someone realizes that the table `toy_table` grows while the system is running and that for some reason there are a lot of elements with atom's as keys. You had expected only integer keys and so does the rest of the system. Well, obviously not all of the system. You turn on call tracing and try to see calls to your module with an atom as the key:

```
1> dbg:tracer().
{ok,<0.88.0>}
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tpl(toy,store,dbg:fun2ms(fun([A,_]) when is_atom(A) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

We use dbg:tpl here to make sure to catch local calls (let's say the module has grown since the smaller version and we're not sure this inserting of atoms is not done locally...). When in doubt always use local call tracing.

Let's say nothing happens when we trace in this way. Our function is never called with these parameters. We make the conclusion that someone else (some other module) is doing it and we realize that we must trace on ets:insert and want to see the calling function. The calling function may be retrieved using the match specification function caller and to get it into the trace message, one has to use the match spec function message. The filter call looks like this (looking for calls to ets:insert):

```
4> dbg:tpl(ets,insert,dbg:fun2ms(fun([toy_table,{A,_}]) when is_atom(A) ->
                                     message(caller())
                                   end)).
{ok,[{matched,nonode@nohost,1},{saved,2}]}
```

The caller will now appear in the “additional message” part of the trace output, and so after a while, the following output comes:

```
(<0.86.0>) call ets:insert(toy_table,{garbage,can}) ({evil_mod,evil_fun,2})
```

You have found out that the function evil_fun of the module evil_mod, with arity 2, is the one causing all this trouble.

This was just a toy example, but it illustrated the most used calls in match specifications for dbg The other, more esotherical calls are listed and explained in the *Users guide of the ERTS application*, they really are beyond the scope of this document.

To end this chatty introduction with something more precise, here follows some parts about caveats and restrictions concerning the fun's used in conjunction with ets:fun2ms and dbg:fun2ms:

Warning:

To use the pseudo functions triggering the translation, one *has to* include the header file ms_transform.hrl in the source code. Failure to do so will possibly result in runtime errors rather than compile time, as the expression may be valid as a plain Erlang program without translation.

Warning:

The fun has to be literally constructed inside the parameter list to the pseudo functions. The fun cannot be bound to a variable first and then passed to ets:fun2ms or dbg:fun2ms, i.e this will work: ets:fun2ms(fun(A) -> A end) but not this: F = fun(A) -> A end, ets:fun2ms(F). The later will result in a compile time error if the header is included, otherwise a runtime error. Even if the later construction would ever appear to work, it really doesn't, so don't ever use it.

Several restrictions apply to the fun that is being translated into a match_spec. To put it simple you cannot use anything in the fun that you cannot use in a match_spec. This means that, among others, the following restrictions apply to the fun itself:

- Functions written in Erlang cannot be called, neither local functions, global functions or real fun's

- Everything that is written as a function call will be translated into a match_spec call to a builtin function, so that the call `is_list(X)` will be translated to `{'is_list', '$1'}` (`'$1'` is just an example, the numbering may vary). If one tries to call a function that is not a match_spec builtin, it will cause an error.
- Variables occurring in the head of the `fun` will be replaced by match_spec variables in the order of occurrence, so that the fragment `fun({A,B,C})` will be replaced by `{'$1', '$2', '$3'}` etc. Every occurrence of such a variable later in the match_spec will be replaced by a match_spec variable in the same way, so that the fun `fun({A,B}) when is_atom(A) -> B end` will be translated into `[{{'$1','$2'},[{is_atom,'$1'}],['$2']}]`.
- Variables that are not appearing in the head are imported from the environment and made into match_spec `const` expressions. Example from the shell:

```
1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
[{{'$1','$2'},[{'>','$1',{const,25}}],['$2']}]
```

- Matching with = cannot be used in the body. It can only be used on the top level in the head of the fun. Example from the shell again:

```
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
[{{'$1',['$2'|'$3']},[{'>','$1','$2'}],['$_']}]
2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
Error: fun with head matching ('=' in head) cannot be translated into
match_spec
{error,transform_error}
3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec
{error,transform_error}
```

  All variables are bound in the head of a match_spec, so the translator can not allow multiple bindings. The special case when matching is done on the top level makes the variable bind to `'$_'` in the resulting match_spec, it is to allow a more natural access to the whole matched object. The pseudo function `object()` could be used instead, see below. The following expressions are translated equally:

```
ets:fun2ms(fun({a,_} = A) -> A end).
ets:fun2ms(fun({a,_}) -> object() end).
```

- The special match_spec variables `'$_'` and `'$*'` can be accessed through the pseudo functions `object()` (for `'$_'`) and `bindings()` (for `'$*'`). as an example, one could translate the following `ets:match_object/2` call to a `ets:select` call:

```
ets:match_object(Table, {'$1',test,'$2'}).
```

  ...is the same as...

```
ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).
```

  (This was just an example, in this simple case the former expression is probably preferable in terms of readability). The `ets:select/2` call will conceptually look like this in the resulting code:

```
ets:select(Table, [{{'$1',test,'$2'},[],['$_']}]).
```

  Matching on the top level of the fun head might feel like a more natural way to access `'$_'`, see above.

- Term constructions/literals are translated as much as is needed to get them into valid match_specs, so that tuples are made into match_spec tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element etc. The guard test `is_record/2` is translated into match_spec code using the three parameter version that's built into match_specs, so that `is_record(A,t)` is translated into `{is_record,'$1',t,5}` given that the record size of record type `t` is 5. Records are of course still not accessible from the shell...
- Language constructions like `case`, `if`, `catch` etc that are not present in match_specs are not allowed.
- The old names for the guard type tests (`list`, `integer`, `float` etc) are not allowed. All guard tests must be written with the new names `is_list`, `is_tuple`, `is_record` etc.
- If the header file `ms_transform.hrl` is not included, the fun won't be translated, which may result in a *runtime error* (depending on if the fun is valid in a pure Erlang context). Be absolutely sure that the header is included when using `ets` and `dbg:fun2ms/1` in compiled code.
- If the pseudo function triggering the translation is `ets:fun2ms/1`, the fun's head must contain a single variable or a single tuple. If the pseudo function is `dbg:fun2ms/1` the fun's head must contain a single variable or a single list.

The translation from fun's to match_specs is done at compile time, so runtime performance is not affected by using these pseudo functions. The compile time might be somewhat longer though.

For more information about match_specs, please read about them in *ERTS users guide*.

Exports

`parse_transform(Forms,_Options) -> Forms`

Types:

- Forms = Erlang abstract code format, see the erl_parse module description
- _Options = Option list, required but not used

Implements the actual transformation at compile time. This function is called by the compiler to do the source code transformation if and when the `ms_transform.hrl` header file is included in your source code. See the `ets` and `dbg`:`fun2ms/1` function manual pages for documentation on how to use this parse_transform, see the `match_spec` chapter in `ERTS` users guide for a description of match specifications.

`transform_from_shell(Dialect,Clauses,BoundEnvironment) -> term()`

Types:

- Dialect = ets | dbg
- Clauses = Erlang abstract form for a single fun
- BoundEnvironment = [{atom(), term()}, ...], list of variable bindings in the shell environment

Implements the actual transformation when the `fun2ms` functions are called from the shell. In this case the abstract form is for one single fun (parsed by the Erlang shell), and all imported variables should be in the key-value list passed ass `BoundEnvironment`. The result is a term, normalized, i.e. not in abstract format.

`format_error(Errcode) -> ErrMessage`

Types:

- Errcode = term()
- ErrMessage = string()

Takes an error code returned by one of the other functions in the module and creates a textual description of the error. Fairly uninteresting function actually.

orddict

Erlang Module

`Orddict` implements a `Key - Value` dictionary. An `orddict` is a representation of a dictionary, where a list of pairs is used to store the keys and values. The list is ordered after the keys.

This module provides exactly the same interface as the module `dict` but with a defined representation.

See Also

dict(3) [page 77], gb_trees(3) [page 151]

ordsets

Erlang 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.

This module provides exactly the same interface as the module `sets` but with a defined representation.

See Also

gb_sets(3) [page 147], sets(3) [page 247]

pg

Erlang 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

Erlang 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 runtime 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 runtime 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

Erlang 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(Fun) -> Pid`
`spawn(Node,Fun) -> Pid`
`spawn(Module,Func,Args) -> Pid`
`spawn(Node,Module,Func,Args) -> Pid`

Types:

- Fun = fun() -> void()
- 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` BIFs. The process can be spawned on another `Node`.

`spawn_link(Fun) -> Pid`
`spawn_link(Node,Fun) -> Pid`
`spawn_link(Module,Func,Args) -> Pid`
`spawn_link(Node,Module,Func,Args) -> Pid`

Types:

- Fun = fun() -> void()
- 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` BIFs. The process can be spawned on another `Node`.

`spawn_opt(Fun,Opts) -> Pid`
`spawn_opt(Node,Fun,Opts) -> Pid`
`spawn_opt(Module,Func,Args,Opts) -> Pid`
`spawn_opt(Node,Module,Func,Args,Opts) -> Pid`

Types:

- Fun = fun() -> void()
- Module = atom()
- Func = atom()
- Args = [Arg]
- Arg = term()
- Node = atom()
- Opts = list()
- Pid = pid()

Spawns a new process and initializes it as described above. The process is spawned using the `spawn_opt` BIFs. The process can be spawned on another `Node`.

`start(Module,Func,Args) -> Ret`
`start(Module,Func,Args,Time) -> Ret`
`start(Module,Func,Args,Time,SpawnOpts) -> Ret`
`start_link(Module,Func,Args) -> Ret`
`start_link(Module,Func,Args,Time) -> Ret`
`start_link(Module,Func,Args,Time,SpawnOpts) -> Ret`

Types:

- Module = atom()
- Func = atom()
- Args = [Arg]
- Arg = term()
- Time = integer >= 0 | infinity
- SpawnOpts = list()
- 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.

The `SpawnOpts` argument, if given, will be passed as the last argument to the `spawn_opt/4` BIF. Refer to the `erlang` module for information about the `spawn_opt` options.

`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} | Fun | false`

Types:

- PidOrPinfo = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = [void()]
- Module = atom()
- Fun = fun() -> void()
- 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 an `erlang:process_info/1` function call).

`translate_initial_call(PidOrPinfo) -> {Module,Function,Arity} | Fun`

Types:

- PidOrPinfo = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = [void()]
- Module = atom()
- Fun = fun() -> void()
- 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 an `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.

`hibernate(Module, Function, Arguments)`

Types:

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

`hibernate/3` gives a way to put a process started using one of the functions in the `proc_lib` module into a wait state where its memory allocation has been reduced as much as possible, which is useful if the process does not expect to receive any messages in the near future.

The process will be awaken when a message is sent to it, and control will resume in `Module:Function` with the arguments given by `ArgumentList`.

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

Note: The actual work is done by the `erlang:hibernate/3` BIF. To ensure that exception handling and logging continues to work in a process started by `proc_lib`, always use `proc_lib:hibernate` rather than `erlang:hibernate/3`.

See Also

error_logger(3)

proplists

Erlang Module

Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples {Atom, true}. (Other terms are allowed in the lists, but are ignored by this module.) If there is more than one entry in a list for a certain key, the first occurrence normally overrides any later (irrespective of the arity of the tuples).

Property lists are useful for representing inherited properties, such as options passed to a function where a user may specify options overriding the default settings, object properties, annotations, etc.

Exports

`append_values(Key, List) -> List`

Types:

- Key = term()
- List = [term()]

Similar to `get_all_values/2`, but each value is wrapped in a list unless it is already itself a list, and the resulting list of lists is concatenated. This is often useful for "incremental" options; e.g., `append_values(a, [{a, [1,2]}, {b, 0}, {a, 3}, {c, -1}, {a, [4]}])` will return the list `[1,2,3,4]`.

`compact(List) -> List`

Types:

- List = [term()]

Minimizes the representation of all entries in the list. This is equivalent to `[property(P) || P <- List]`.

See also: `property/1`, `unfold/1`.

`delete(Key, List) -> List`

Types:

- Key = term()
- List = [term()]

Deletes all entries associated with `Key` from `List`.

`expand(Expansions, List) -> List`

Types:

- Key = term()
- Expansions = [{Property,[term()]}]
- Property = atom() | tuple()

Expands particular properties to corresponding sets of properties (or other terms). For each pair {`Property, Expansion`} in `Expansions`, if `E` is the first entry in `List` with the same key as `Property`, and `E` and `Property` have equivalent normal forms, then `E` is replaced with the terms in `Expansion`, and any following entries with the same key are deleted from `List`.

For example, the following expressions all return `[fie, bar, baz, fum]`:

```
expand([{foo, [bar, baz]}],
[fie, foo, fum])
expand([{{foo, true}, [bar, baz]}],
[fie, foo, fum])
expand([{{foo, false}, [bar, baz]}],
[fie, {foo, false}, fum])
```

However, no expansion is done in the following call:

```
expand([{{foo, true}, [bar, baz]}],
[{foo, false}, fie, foo, fum])
```

because {`foo, false`} shadows `foo`.

Note that if the original property term is to be preserved in the result when expanded, it must be included in the expansion list. The inserted terms are not expanded recursively. If `Expansions` contains more than one property with the same key, only the first occurrance is used.

See also: `normalize/2`.

`get_all_values(Key, List) -> [term()]`

Types:

- Key = term()
- List = [term()]

Similar to `get_value/2`, but returns the list of values for *all* entries {`Key, Value`} in `List`. If no such entry exists, the result is the empty list.

See also: `get_value/2`.

`get_bool(Key, List) -> bool()`

Types:

- Key = term()
- List = [term()]

Returns the value of a boolean key/value option. If `lookup(Key, List)` would yield {`Key, true`}, this function returns `true`; otherwise `false` is returned.

See also: `get_value/2`, `lookup/2`.

`get_keys(List) -> [term()]`

Types:

- List = [term()]

Returns an unordered list of the keys used in `List`, not containing duplicates.

`get_value(Key, List) -> term()`

Types:

- Key = term()
- List = [term()]

Equivalent to `get_value(Key, List, undefined)`.

`get_value(Key, List, Default) -> term()`

Types:

- Key = term()
- Default = term()
- List = [term()]

Returns the value of a simple key/value property in `List`. If `lookup(Key, List)` would yield `{Key, Value}`, this function returns the corresponding `Value`, otherwise `Default` is returned.

See also: `get_all_values/2`, `get_bool/2`, `get_value/1`, `lookup/2`.

`is_defined(Key, List) -> bool()`

Types:

- Key = term()
- List = [term()]

Returns `true` if `List` contains at least one entry associated with `Key`, otherwise `false` is returned.

`lookup(Key, List) -> none | tuple()`

Types:

- Key = term()
- List = [term()]

Returns the first entry associated with `Key` in `List`, if one exists, otherwise returns `none`. For an atom `A` in the list, the tuple `{A, true}` is the entry associated with `A`.

See also: `get_bool/2`, `get_value/2`, `lookup_all/2`.

`lookup_all(Key, List) -> [tuple()]`

Types:

- Key = term()
- List = [term()]

Returns the list of all entries associated with `Key` in `List`. If no such entry exists, the result is the empty list.

See also: `lookup/2`.

`normalize(List, Stages) -> List`

Types:

- List = [term()]
- Stages = [Operation]
- Operation = {aliases, Aliases} | {negations, Negations} | {expand, Expansions}
- Aliases = [{Key, Key}]
- Negations = [{Key, Key}]
- Key = term()
- Expansions = [{Property, [term()]}]
- Property = atom() | tuple()

Passes `List` through a sequence of substitution/expansion stages. For an `aliases` operation, the function `substitute_aliases/2` is applied using the given list of aliases; for a `negations` operation, `substitute_negations/2` is applied using the given negation list; for an `expand` operation, the function `expand/2` is applied using the given list of expansions. The final result is automatically compacted (cf. `compact/1`).

Typically you want to substitute negations first, then aliases, then perform one or more expansions (sometimes you want to pre-expand particular entries before doing the main expansion). You might want to substitute negations and/or aliases repeatedly, to allow such forms in the right-hand side of aliases and expansion lists.

See also: `compact/1`, `expand/2`, `substitute_aliases/2`, `substitute_negations/2`.

`property(Property) -> Property`

Types:

- Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If `Property` is `{Key, true}` where `Key` is an atom, this returns `Key`, otherwise the whole term `Property` is returned.

See also: `property/2`.

`property(Key, Value) -> Property`

Types:

- Key = term()
- Value = term()
- Property = atom() | tuple()

Creates a normal form (minimal) representation of a simple key/value property. Returns `Key` if `Value` is `true` and `Key` is an atom, otherwise a tuple `{Key, Value}` is returned.

See also: `property/1`.

`split(List, Keys) -> {Lists, Rest}`

Types:

- List = [term()]
- Keys = [term()]
- Lists = [[term()]]
- Rest = [term()]

Partitions `List` into a list of sublists and a remainder. `Lists` contains one sublist for each key in `Keys`, in the corresponding order. The relative order of the elements in each sublist is preserved from the original `List`. `Rest` contains the elements in `List` that are not associated with any of the given keys, also with their original relative order preserved.

Example: split([{c, 2}, {e, 1}, a, {c, 3, 4}, d, {b, 5}, b], [a, b, c])

returns

{[[a], [{b, 5}, b],[{c, 2}, {c, 3, 4}]], [{e, 1}, d]}

`substitute_aliases(Aliases, List) -> List`

Types:

- Aliases = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of properties. For each entry in `List`, if it is associated with some key `K1` such that `{K1, K2}` occurs in `Aliases`, the key of the entry is changed to `Key2`. If the same `K1` occurs more than once in `Aliases`, only the first occurrence is used.

Example: `substitute_aliases([{color, colour}], L)` will replace all tuples `{color, ...}` in `L` with `{colour, ...}`, and all atoms `color` with `colour`.

See also: `normalize/2`, `substitute_negations/2`.

`substitute_negations(Negations, List) -> List`

Types:

- Negations = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of boolean-valued properties and simultaneously negates their values. For each entry in `List`, if it is associated with some key `K1` such that `{K1, K2}` occurs in `Negations`, then if the entry was `{K1, true}` it will be replaced with `{K2, false}`, otherwise it will be replaced with `{K2, true}`, thus changing the name of the option and simultaneously negating the value given by `get_bool(List)`. If the same `K1` occurs more than once in `Negations`, only the first occurrence is used.

Example: `substitute_negations([{no_foo, foo}], L)` will replace any atom `no_foo` or tuple `{no_foo, true}` in `L` with `{foo, false}`, and any other tuple `{no_foo, ...}` with `{foo, true}`.

See also: `get_bool/2`, `normalize/2`, `substitute_aliases/2`.

`unfold(List) -> List`

Types:

- List = [term()]

Unfolds all occurences of atoms in `List` to tuples `{Atom, true}`.

queue

Erlang Module

This module implements FIFO queues in an efficient manner.

All operations has an amortised O(1) running time, except `len/1`, `reverse/1`, `join/2` and `split/2` that probably are O(n).

Exports

`cons(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the head of queue `Q1`. Returns the new queue `Q2`.

`daeh(Q) -> Item`

The same as `last(Q)` and the opposite of `head(Q)`.

`from_list(L) -> queue()`

Types:

- L = list()

Returns a queue containing the items in `L`, in the same order - the head item of the list will be the head item of the queue.

`head(Q) -> Item`

Types:

- Item = term()
- Q = queue()

Returns `Item` from the head of queue `Q`.

Fails with reason `empty` if `Q` is empty.

`in(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the tail of queue `Q1`. Returns a new queue `Q2`. This is the same as `snoc(Q1, Item)`.

`in_r(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the head of queue `Q1`. Returns a new queue `Q2`. This is the same as `cons(Item, Q1)`.

`init(Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the last item from `Q1`. This is the opposite of `tail(Q1)`.

Fails with reason `empty` if `Q1` is empty.

`is_empty(Q) -> true | false`

Types:

- Q = queue()

Tests if `Q` is empty and returns `true` if so and `false` otherwise.

`join(Q1, Q2) -> Q3`

Types:

- Q1 = Q2 = Q3 = queue()

Returns a queue `Q3` that is the result of joining `Q1` and `Q2` with `Q1` before (at the head) `Q2`.

`lait(Q1) -> Q2`

The same as `init(Q1)` and the opposite of `tail(Q1)`.

`last(Q) -> Item`

Types:

- Item = term()
- Q = queue()

Returns the last item of queue `Q`. This is the opposite of `head(Q)`.

Fails with reason `empty` if `Q` is empty.

`len(Q) -> N`

Types:

- Q = queue()
- N = integer()

Calculates and returns the length of queue `Q`.

`new() -> Q`

Types:

- Q = queue()

Returns an empty queue.

`out(Q1) -> Result`

Types:

- Result = {{value, Item}, Q2} | {empty, Q1}
- Q1 = Q2 = queue()

Removes the head item from the queue `Q1`. Returns the tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the new queue. If `Q1` is empty, the tuple `{empty, Q1}` is returned.

`out_r(Q1) -> Result`

Types:

- Result = {{value, Item}, Q2} | {empty, Q1}
- Q1 = Q2 = queue()

Removes the last item from the queue `Q1`. Returns the tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the new queue. If `Q1` is empty, the tuple `{empty, Q1}` is returned.

`reverse(Q1) -> Q2`

Types:

- Q1 = Q2 = queue()

Returns a queue `Q2` that contains the items of `Q1` in the reverse order.

`snoc(Q1, Item) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` as the last item of queue `Q1`. Returns the new queue `Q2`. This is the opposite of `cons(Item, Q1)`.

`split(N, Q1) -> {Q2,Q3}`

Types:

- N = integer()
- Q1 = Q2 = Q3 = queue()

Splits `Q1` into a queue `Q2` of length `N` with items from the head end, and a queue `Q3` with the rest of the items.

`tail(Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the head item from `Q1`.

Fails with reason `empty` if `Q1` is empty.

`to_list(Q) -> list()`

Types:

- Q = queue()

Returns a list of the items in the queue, with the head item of the queue as the head of the list.

random

Erlang 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.

Every time a random number is requested, a state is used to calculate it, and a new state produced. The state can either be implicit (kept in the process dictionary) or be an explicit argument and return value. In this implementation, the state (the type `ran()`) consists of a tuple of three integers.

Exports

`seed() -> ran()`

Seeds random number generation with default (fixed) values in the process dictionary, and returns the old state.

`seed(A1, A2, A3) -> ran()`

Types:

- A1 = A2 = A3 = int()

Seeds random number generation with integer values in the process dictionary, and returns the old state.

`seed0() -> ran()`

Returns the default state.

`uniform()-> float()`

Returns a random float uniformly distributed between `0.0` and `1.0`, updating the state in the process dictionary.

`uniform(N) -> int()`

Types:

- N = int()

Given an integer `N >= 1`, `uniform/1` returns a random integer uniformly distributed between `1` and `N`, updating the state in the process dictionary.

`uniform_s(State0) -> {float(), State1}`

Types:

- State0 = State1 = ran()

Given a state, `uniform_s/1`returns a random float uniformly distributed between `0.0` and `1.0`, and a new state.

`uniform_s(N, State0) -> {int(), State1}`

Types:

- N = int()
- State0 = State1 = ran()

Given an integer `N >= 1` and a state, `uniform_s/2` returns a random integer uniformly distributed between `1` and `N`, and a new state.

Note

Some of the functions use the process dictionary variable `random_seed` to remember the current seed.

If a process calls `uniform/0` or `uniform/1` without setting a seed first, `seed/0` is called automatically.

regexp

Erlang 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 rs.

r* matches zero or more rs.

r? matches zero or one rs.

(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.

sets

Erlang Module

Sets are collections of elements with no duplicate elements. The representation of a set is not defined.

Exports

`new() -> Set`

Types:

- Set = set()

Returns a new empty ordered set.

`is_set(Set) -> bool()`

Types:

- Set = term()

Returns `true` if `Set` is an ordered set of elements, otherwise `false`.

`size(Set) -> int()`

Types:

- Set = term()

Returns the number of elements in `Set`.

`to_list(Set) -> List`

Types:

- Set = set()
- List = [term()]

Returns the elements of `Set` as a list.

`from_list(List) -> Set`

Types:

- List = [term()]
- Set = set()

Returns an ordered set of the elements in `List`.

`is_element(Element, Set) -> bool()`

Types:

- Element = term()
- Set = set()

Returns `true` if `Element` is an element of `Set`, otherwise `false`.

`add_element(Element, Set1) -> Set2`

Types:

- Element = term()
- Set1 = Set2 = set()

Returns a new ordered set formed from `Set1` with `Element` inserted.

`del_element(Element, Set1) -> Set2`

Types:

- Element = term()
- Set1 = Set2 = set()

Returns `Set1`, but with `Element` removed.

`union(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the merged (union) set of `Set1` and `Set2`.

`union(SetList) -> Set`

Types:

- SetList = [set()]
- Set = set()

Returns the merged (union) set of the list of sets.

`intersection(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the intersection of `Set1` and `Set2`.

`intersection(SetList) -> Set`

Types:

- SetList = [set()]
- Set = set()

Returns the intersection of the non-empty list of sets.

`subtract(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns only the elements of `Set1` which are not also elements of `Set2`.

`is_subset(Set1, Set2) -> bool()`

Types:

- Set1 = Set2 = set()

Returns `true` when every element of `Set1` is also a member of `Set2`, otherwise `false`.

`fold(Function, Acc0, Set) -> Acc1`

Types:

- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Set = set()

Fold `Function` over every element in `Set` returning the final value of the accumulator.

`filter(Pred, Set1) -> Set2`

Types:

- Pred = fun (E) -> bool()
- Set1 = Set2 = set()

Filter elements in `Set1` with boolean function `Fun`.

See Also

ordsets(3) [page 222], gb_sets(3) [page 147]

shell

Erlang 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. How many commands and results to save can be determined by the user, either interactively, by calling shell:history/1 and shell:results/1, or by setting the application configuration parameters shell_history_length and shell_saved_results for the application stdlib.

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 (possibly restricted) 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.

`history(N)` Sets the number of previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`results(N)` Sets the number of results from previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`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 5.3 [hipe] [threads:0]

Eshell V5.3  (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).
** 1: variable 'L' is unbound **
```

```
9> {L, _} = Descriptor.
{4,abcd}
10> L.
4
11> {P, Q, R} = Descriptor.
** exited: {{badmatch,{4,abcd}},[{erl_eval,expr,3}]} **
12> P.
** 1: variable 'P' is unbound **
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).

=ERROR REPORT==== 19-Feb-2003::10:04:14 ===
Error in process <0.40.0> with exit value: {{badmatch,1},[{test1,demo,1},
{erl_eval,expr,4},{shell,eval_loop,2}]}
** exited: {{badmatch,1},
            [{test1,demo,1},{erl_eval,expr,4},{shell,eval_loop,2}]} **
23> Z.
** 1: variable 'Z' is unbound **
24> get(aa).
hello
25> erase(), put(aa, hello).
undefined
26> spawn(test1, demo, [1]).
<0.57.0>
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 `L` is no longer bound to a value.

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 runtime 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.

The shell escape key `^G` (Control G) detaches the current job and activates `JCL` mode. 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. Note that this option is disabled if Erlang is started with the ignore break, `+Bi`, system flag (which may be useful e.g. when running a restricted shell, see below).

`?` Displays this message.

It is possible to alter the behaviour of shell escape by means of the `stdlib` application variable `shell_esc`. The value of the variable can be either `jcl` (`erl -stdlib shell_esc jcl`) or `abort` (`erl -stdlib shell_esc abort`). The first option sets ^G to activate `JCL` mode (which is also default behaviour). The latter sets ^G to terminate the current shell and start a new one. `JCL` mode can not be invoked when `shell_esc` is set to `abort`.

Restricted Shell

The shell may be started in a restricted mode. In this mode, the shell evaluates a function call only if allowed. This feature makes it possible to, for example, prevent a user from accidentally calling a function from the prompt that could harm a running system (useful in combination with the the system flag `+Bi`).

When the restricted shell evaluates an expression and encounters a function call, it calls a predicate function (with information about the function call in question). This predicate function returns `true` to let the shell go ahead with the evaluation, or `false` to abort it. There are two possible predicate functions for the user to implement:

`local_allowed(Func, ArgList, State) -> {true,NewState} | {false,NewState}`

to determine if the call to the local function `Func` with arguments `ArgList` should be allowed.

`non_local_allowed(FuncSpec, ArgList, State) -> {true,NewState} | {false,NewState}`

to determine if the call to non-local function `FuncSpec` (`{Module,Func}` or a fun) with arguments `ArgList` should be allowed.

These predicate functions are in fact called from local and non-local evaluation function handlers, described in the erl_eval [page 94] manual page. (Arguments in `ArgList` are evaluated before the predicates are called).

The `State` argument is a tuple `{ShellState,ExprState}`. The return value `NewState` has the same form. This may be used to carry a state between calls to the predicate functions. Data saved in `ShellState` lives through an entire shell session. Data saved in `ExprState` lives only through the evaluation of the current expression.

There are two ways to start a restricted shell session:

- Use the stdlib application variable `restricted_shell` and specify, as its value, the name of the predicate function module. Example (with predicate functions implemented in pred_mod.erl): `$ erl -stdlib restricted_shell pred_mod`
- From a normal shell session, call function `shell:start_restricted/1`. This exits the current evaluator and starts a new one in restricted mode.

Notes:

- When restricted shell mode is activated or deactivated, new jobs started on the node will run in restricted or normal mode respectively.
- If restricted mode has been enabled on a particular node, remote shells connecting to this node will also run in restricted mode.
- The predicate functions can not be used to allow or disallow execution of functions called from compiled code (only functions called from expressions entered at the shell prompt).

Exports

`history(N) -> integer()`

Types:

- N = integer()

Sets the number of previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`results(N) -> integer()`

Types:

- N = integer()

Sets the number of results from previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`start_restricted(Module) -> ok`

Types:

- Module = atom()

Exits a normal shell and starts a restricted shell. `Module` specifies the module for the predicate functions `local_allowed/3` and `non_local_allowed/3`. The function is meant to be called from the shell.

`stop_restricted() -> ok`

Exits a restricted shell and starts a normal shell. The function is meant to be called from the shell.

shell_default

Erlang 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").
"cba"
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

Erlang 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, Erlang/OTP 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:

- Erlang/OTP is not installed on the remote host
- the file system on the other host has a different structure to the the master
- the Erlang nodes 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 runtime 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 pseudo 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`.

sofs

Erlang Module

The `sofs` module implements operations on finite sets and relations represented as sets. Intuitively, a set is a collection of elements; every element belongs to the set, and the set contains every element.

Given a set A and a sentence S(x), where x is a free variable, a new set B whose elements are exactly those elements of A for which S(x) holds can be formed, this is denoted B= {xinA: S(x)}. Sentences are expressed using the logical operators "for some" (or "there exists"), "for all", "and", "or", "not". If the existence of a set containing all the specified elements is known (as will always be the case in this module), we write B= {x: S(x)}.

The *unordered set* containing the elements a, b and c is denoted {a,b,c}. This notation is not to be confused with tuples. The *ordered pair* of a and b, with first *coordinate* a and second coordinate b, is denoted (a,b). An ordered pair is an *ordered set* of two elements. In this module ordered sets can contain one, two or more elements, and parentheses are used to enclose the elements. Unordered sets and ordered sets are orthogonal, again in this module; there is no unordered set equal to any ordered set.

The set that contains no elements is called the *empty set*. If two sets A and B contain the same elements, then A is *equal* to B, denoted A=B. Two ordered sets are equal if they contain the same number of elements and have equal elements at each coordinate. If a set A contains all elements that B contains, then B is a *subset* of A. The *union* of two sets A and B is the smallest set that contains all elements of A and all elements of B. The *intersection* of two sets A and B is the set that contains all elements of A that belong to B. Two sets are *disjoint* if their intersection is the empty set. The *difference* of two sets A and B is the set that contains all elements of A that do not belong to B. The *symmetric difference* of two sets is the set that contains those element that belong to either of the two sets, but not both. The *union* of a collection of sets is the smallest set that contains all the elements that belong to at least one set of the collection. The *intersection* of a non-empty collection of sets is the set that contains all elements that belong to every set of the collection.

The *Cartesian product* of two sets X and Y, denoted XY, is the set {a: a= (x,y) for some xinX and for some yinY}. A *relation* is a subset of XY. Let R be a relation. The fact that (x,y) belongs to R is written as xRy. Since relations are sets, the definitions of the last paragraph (subset, union, and so on) apply to relations as well. The *domain* of R is the set {x: xRy for some yinY}. The *range* of R is the set {y: xRy for some xinX}. The *converse* of R is the set {a: a= (y,x) for some (x,y)inR}. If A is a subset of X, then the *image* of A under R is the set {y: xRy for some xinA}, and if B is a subset of Y, then the *inverse image* of B is the set {x: xRy for some yinB}. If R is a relation from X to Y and S is a relation from Y to Z, then the *relative product* of R and S is the relation T from X to Z defined so that xTz if and only if there exists an element y in Y such that xRy and ySz. The *restriction* of R to A is the set S defined so that xSy if and only if there exists an element x in A such that xRy. If S is a restriction of R to A, then R is an *extension* of S to X. If X=Y then we call R a relation *in* X. The *field* of a relation R in X is the union of the domain of R and the range of R. If R is a relation in X, and if S is defined so that xSy if xRy and not x=y, then S is the *strict* relation corresponding to R, and vice versa, if S is a relation in X, and if R is defined so that xRy if xSy or x=y, then R is the *weak* relation corresponding to S. A relation R in X is *reflexive* if xRx for every element x of X; it is *symmetric* if xRy implies that yRx; and it is *transitive* if xRy and yRz imply that xRz.

A *function* F is a relation, a subset of XY, such that the domain of F is equal to X and such that for every x in X there is a unique element y in Y with (x,y) in F. The latter condition can be formulated as follows: if xFy and xFz then y=z. In this module, it will not be required that the domain of F be equal to X for a relation to be considered a function. Instead of writing (x,y)inF or xFy, we write F(x)=y when F is a function, and say that F maps x onto y, or that the value of F at x is y. Since functions are relations, the definitions of the last paragraph (domain, range, and so on) apply to functions as well. If the converse of a function F is a function F', then F' is called the *inverse* of F. The relative product of two functions F1 and F2 is called the *composite* of F1 and F2 if the range of F1 is a subset of the domain of F2.

Sometimes, when the range of a function is more important than the function itself, the function is called a *family*. The domain of a family is called the *index set*, and the range is called the *indexed set*. If x is a family from I to X, then x[i] denotes the value of the function at index i. The notation "a family in X" is used for such a family. When the indexed set is a set of subsets of a set X, then we call x a *family of subsets* of X. If x is a family of subsets of X, then the union of the range of x is called the *union of the family* x. If x is non-empty (the index set is non-empty), the *intersection of the family* x is the intersection of the range of x. In this module, the only families that will be considered are families of subsets of some set X; in the following the word "family" will be used for such families of subsets.

A *partition* of a set X is a collection S of non-empty subsets of X whose union is X and whose elements are pairwise disjoint. A relation in a set is an *equivalence relation* if it is reflexive, symmetric and transitive. If R is an equivalence relation in X, and x is an element of X, the *equivalence class* of x with respect to R is the set of all those elements y of X for which xRy holds. The equivalence classes constitute a partitioning of X. Conversely, if C is a partition of X, then the relation that holds for any two elements of X if they belong to the same equivalence class, is an equivalence relation induced by the partition C. If R is an equivalence relation in X, then the *canonical map* is the function that maps every element of X onto its equivalence class.

Relations as defined above (as sets of ordered pairs) will from now on be referred to as *binary relations*. We call a set of ordered sets (x[1],...,x[n]) an *(n-ary) relation*, and say that the relation is a subset of the Cartesian product X[1]...X[n] where x[i] is an element of X[i], 1<=i<=n. The *projection* of an n-ary relation R onto coordinate i is the set {x[i]: (x[1],...,x[i],...,x[n]) in R for some x[j]inX[j], 1<=j<=n and not i=j}. The projections of a binary relation R onto the first and second coordinates are the domain and the range of R respectively. The relative product of binary relations can be generalized to n-ary relations as follows. Let TR be an ordered set (R[1],...,R[n]) of binary relations from X to Y[i] and S a binary relation from (Y[1]...Y[n]) to Z. The *relative product* of TR and S is the binary relation T from X to Z defined so that xTz if and only if there exists an element y[i] in Y[i] for each 1<=i<=n such that xR[i]y[i] and (y[1],...,y[n])Sz. Now let TR be a an ordered set (R[1],...,R[n]) of binary relations from X[i] to Y[i] and S a subset of X[1]...X[n]. The *multiple relative product* of TR and and S is defined to be the set {z: z= ((x[1],...,x[n]), (y[1],...,y[n])) for some (x[1],...,x[n])inS and for some (x[i],y[i]) in R[i], 1<=i<=n}. The *natural join* of an n-ary relation R and an m-ary relation S on coordinate i and j is defined to be the set {z:

z= (x[1],...,x[n], y[1],...,y[j-1],y[j+1],...,y[m]) for some (x[1],...,x[n])inR and for some (y[1],...,y[m])inS such that x[i]=y[j]}.

The sets recognized by this module will be represented by elements of the relation Sets, defined as the smallest set such that:

- for every atom T except '_' and for every term X, (T,X) belongs to Sets (*atomic sets*);
- (['_'],[]) belongs to Sets (the *untyped empty set*);
- for every tuple T= {T[1],...,T[n]} and for every tuple X= {X[1],...,X[n]}, if (T[i],X[i]) belongs to Sets for every 1<=i<=n then (T,X) belongs to Sets (*ordered sets*);
- for every term T, if X is the empty list or a non-empty sorted list [X[1],...,X[n]] without duplicates such that (T,X[i]) belongs to Sets for every 1<=i<=n, then ([T],X) belongs to Sets (*typed unordered sets*).

An *external set* is an element of the range of Sets. A *type* is an element of the domain of Sets. If S is an element (T,X) of Sets, then T is a *valid type* of X, T is the type of S, and X is the external set of S. from_term/2 [page 273] creates a set from a type and an Erlang term turned into an external set.

The actual sets represented by Sets are the elements of the range of the function Set from Sets to Erlang terms and sets of Erlang terms:

- Set(T,Term)= Term, where T is an atom;
- Set({T[1],...,T[n]},{X[1],...,X[n]}) = (Set(T[1],X[1]),...,Set(T[n],X[n]));
- Set([T],[X[1],...,X[n]]) = {Set(T,X[1]),...,Set(T,X[n])};
- Set([T],[])= {}.

When there is no risk of confusion, elements of Sets will be identified with the sets they represent. For instance, if U is the result of calling `union/2` with S1 and S2 as arguments, then U is said to be the union of S1 and S2. A more precise formulation would be that Set(U) is the union of Set(S1) and Set(S2).

The types are used to implement the various conditions that sets need to fulfill. As an example, consider the relative product of two sets R and S, and recall that the relative product of R and S is defined if R is a binary relation to Y and S is a binary relation from Y. The function that implements the relative product, relative_product/2 [page 280], checks that the arguments represent binary relations by matching [{A,B}] against the type of the first argument (Arg1 say), and [{C,D}] against the type of the second argument (Arg2 say). The fact that [{A,B}] matches the type of Arg1 is to be interpreted as Arg1 representing a binary relation from X to Y, where X is defined as all sets Set(x) for some element x in Sets the type of which is A, and similarly for Y. In the same way Arg2 is interpreted as representing a binary relation from W to Z. Finally it is checked that B matches C, which is sufficient to ensure that W is equal to Y. The untyped empty set is handled separately: its type, ['_'], matches the type of any unordered set.

A few functions of this module (`drestriction/3`, `family_projection/2`, `partition/2`, `partition_family/2`, `projection/2`, `restriction/3`, `substitution/2`) accept an Erlang function as a means to modify each element of a given unordered set. Such a function, called SetFun in the following, can be specified as a functional object (fun), a tuple {`external,Fun`}, or an integer. If SetFun is specified as a fun, the fun is applied to each element of the given set and the return value is assumed to be a set. If SetFun is specified as a tuple {external, Fun}, Fun is applied to the external set of each element of the given set and the return value is assumed to be an external set. Selecting the elements of an unordered set as external sets and assembling a new unordered set from a list of external sets is in the present implementation more efficient than modifying each element as a set. However, this optimization can only be utilized when the elements of the unordered set are atomic or ordered sets. It must also be the case that the type of the elements matches some clause of Fun (the type of the created set is the result of applying Fun to the type of the given set), and that Fun does nothing but selecting, duplicating or rearranging parts of the elements. Specifying a SetFun as an integer I is equivalent to specifying {external, fun(X)-> element(I,X)}, but is to be preferred since it makes it possible to handle this case even more efficiently. Examples of SetFuns:

```
{sofs, union}
fun(S) -> sofs:partition(1, S) end
{external, fun(A) -> A end}
{external, fun({A,_,C}) -> {C,A} end}
{external, fun({_,{_,C}}) -> C end}
{external, fun({_,{_,{_,E}=C}}) -> {E,{E,C}} end}
2
```

The order in which a SetFun is applied to the elements of an unordered set is not specified, and may change in future versions of sofs.

The execution time of the functions of this module is dominated by the time it takes to sort lists. When no sorting is needed, the execution time is in the worst case proportional to the sum of the sizes of the input arguments and the returned value. A few functions execute in constant time: from_external, is_empty_set, is_set, is_sofs_set, to_external, type.

The functions of this module exit the process with a badarg, bad_function, or type_mismatch message when given badly formed arguments or sets the types of which are not compatible.

Types

```
anyset() = -an unordered, ordered or atomic set-
binary_relation() = -a binary relation-
bool() = true | false
external_set() = -an external set-
family() = -a family (of subsets)-
function() = -a function-
ordset() = -an ordered set-
relation() = -an n-ary relation-
set() = -an unordered set-
set_of_sets() = -an unordered set of set()-
set_fun() = integer() >= 1
          | {external, fun(external_set()) -> external_set()}
          | fun(anyset()) -> anyset()
spec_fun() = {external, fun(external_set()) -> bool()}
           | fun(anyset()) -> bool()
type() = -a type-
```

Exports

`a_function(Tuples [, Type]) -> Function`

Types:

- Function = function()
- Tuples = [tuple()]
- Type = type()

Creates a function [page 263]. `a_function(F,T)` is equivalent to `from_term(F,T)`, if the result is a function. If no type [page 264] is explicitly given, `[{atom,atom}]` is used as type of the function.

`canonical_relation(SetOfSets) -> BinRel`

Types:

- BinRel = binary_relation()
- SetOfSets = set_of_sets()

Returns the binary relation containing the elements (E,Set) such that Set belongs to SetOfSets and E belongs to Set. If SetOfSets is a partition [page 263] of a set X and R is the equivalence relation in X induced by SetOfSets, then the returned relation is the canonical map [page 263] from X onto the equivalence classes with respect to R.

```
1> Ss = sofs:from_term([[a,b],[b,c]]),
CR = sofs:canonical_relation(Ss),
sofs:to_external(CR).
[{a,[a,b]},{b,[a,b]},{b,[b,c]},{c,[b,c]}]
```

`composite(Function1, Function2) -> Function3`

Types:

- Function1 = Function2 = Function3 = function()

Returns the composite [page 263] of the functions Function1 and Function2.

```
1> F1 = sofs:a_function([{a,1},{b,2},{c,2}]),
F2 = sofs:a_function([{1,x},{2,y},{3,z}]),
F = sofs:composite(F1, F2),
sofs:to_external(F).
[{a,x},{b,y},{c,y}]
```

`constant_function(Set, AnySet) -> Function`

Types:

- AnySet = anyset()
- Function = function()
- Set = set()

Creates the function [page 263] that maps each element of the set Set onto AnySet.

```
1> S = sofs:set([a,b]),
E = sofs:from_term(1),
R = sofs:constant_function(S, E),
sofs:to_external(R).
[{a,1},{b,1}]
```

`converse(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns the converse [page 262] of the binary relation BinRel1.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,a}]),
R2 = sofs:converse(R1),
sofs:to_external(R2).
[{a,1},{a,3},{b,2}]
```

`difference(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the difference [page 262] of the sets Set1 and Set2.

`digraph_to_family(Graph [, Type]) -> Family`

Types:

- Graph = digraph() -see digraph(3)-
- Family = family()
- Type = type()

Creates a family [page 263] from the directed graph Graph. Each vertex a of Graph is represented by a pair (a,{b[1],...,b[n]}) where the b[i]'s are the out-neighbours of a. If no type is explicitly given, [{atom,[atom]}] is used as type of the family. It is assumed that Type is a valid type [page 264] of the external set of the family.

If G is a directed graph, it holds that the vertices and edges of G are the same as the vertices and edges of `family_to_digraph(digraph_to_family(G))`.

`domain(BinRel) -> Set`

Types:

- BinRel = binary_relation()
- Set = set()

Returns the domain [page 262] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:domain(R),
sofs:to_external(S).
[1,2]
```

`drestriction(BinRel1, Set) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the difference between the binary relation BinRel1 and the restriction [page 262] of BinRel1 to Set.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
R2 = sofs:drestriction(R1, S),
sofs:to_external(R2).
[{1,a},{3,c}]
```

`drestriction(R,S)` is equivalent to `difference(R,restriction(R,S))`.

`drestriction(SetFun, Set1, Set2) -> Set3`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that do not yield an element in Set2 as the result of applying SetFun.

```
1> SetFun = {external, fun({_A,B,C}) -> {B,C} end},
R1 = sofs:relation([{a,aa,1},{b,bb,2},{c,cc,3}]),
R2 = sofs:relation([{bb,2},{cc,3},{dd,4}]),
R3 = sofs:drestriction(SetFun, R1, R2),
sofs:to_external(R3).
[{a,aa,1}]
```

`drestriction(F,S1,S2)` is equivalent to `difference(S1,restriction(F,S1,S2))`.

`empty_set() -> Set`

Types:

- Set = set()

Returns the untyped empty set [page 264]. `empty_set()` is equivalent to `from_term([],['_'])`.

`extension(BinRel1, Set, AnySet) -> BinRel2`

Types:

- AnySet = anyset()
- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the extension [page 262] of BinRel1 such that for each element E in Set that does not belong to the domain [page 262] of BinRel1, BinRel2 contains the pair (E,AnySet).

```
1> S = sofs:set([b,c]),
A = sofs:empty_set(),
R = sofs:family([{a,[1,2]},{b,[3]}]),
X = sofs:extension(R, S, A),
sofs:to_external(X).
[{a,[1,2]},{b,[3]},{c,[]}]
```

`family(Tuples [, Type]) -> Family`

Types:

- Family = family()
- Tuples = [tuple()]
- Type = type()

Creates a family of subsets [page 263]. `family(F,T)` is equivalent to `from_term(F,T)`, if the result is a family. If no type [page 264] is explicitly given, `[{atom,[atom]}]` is used as type of the family.

`family_difference(Family1, Family2) -> Family3`

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 263], then Family3 the family such that the index set is equal to the index set of Family1, and Family3[i] is the difference between Family1[i] and Family2[i] if Family2 maps i, Family1[i] otherwise.

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]}]),
F2 = sofs:family([{b,[4,5]},{c,[6,7]}]),
F3 = sofs:family_difference(F1, F2),
sofs:to_external(F3).
[{a,[1,2]},{b,[3]}]
```

`family_domain(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 263] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the domain [page 262] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_domain(FR),
sofs:to_external(F).
[{a,[1,2,3]},{b,[]},{c,[4,5]}]
```

`family_field(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 263] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the field [page 262] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_field(FR),
sofs:to_external(F).
[{a,[1,2,3,a,b,c]},{b,[]},{c,[4,5,d,e]}]
```

`family_field(Family1)` is equivalent to `family_union(family_domain(Family1), family_range(Family1))`.

`family_intersection(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 263] and Family1[i] is a set of sets for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the intersection [page 262] of Family1[i].

If Family1[i] is an empty set for some i, then the process exits with a `badarg` message.

```
1> F1 = sofs:from_term([{a,[[1,2,3],[2,3,4]]},{b,[[x,y,z],[x,y]]}]),
F2 = sofs:family_intersection(F1),
sofs:to_external(F2).
[{a,[2,3]},{b,[x,y]}]
```

`family_intersection(Family1, Family2) -> Family3`

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 263], then Family3 is the family such that the index set is the intersection of Family1's and Family2's index sets, and Family3[i] is the intersection of Family1[i] and Family2[i].

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]},{c,[5,6]}]),
F2 = sofs:family([{b,[4,5]},{c,[7,8]},{d,[9,10]}]),
F3 = sofs:family_intersection(F1, F2),
sofs:to_external(F3).
[{b,[4]},{c,[]}]
```

`family_projection(SetFun, Family1) -> Family2`

Types:

- SetFun = set_fun()
- Family1 = Family2 = family()
- Set = set()

If Family1 is a family [page 263] then Family2 is the family with the same index set as Family1 such that Family2[i] is the result of calling SetFun with Family1[i] as argument.

```
1> F1 = sofs:from_term([{a,[[1,2],[2,3]]},{b,[[]]}]),
F2 = sofs:family_projection({sofs, union}, F1),
sofs:to_external(F2).
[{a,[1,2,3]},{b,[]}]
```

`family_range(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 263] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the range [page 262] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_range(FR),
sofs:to_external(F).
[{a,[a,b,c]},{b,[]},{c,[d,e]}]
```

`family_specification(Fun, Family1) -> Family2`

Types:

- Fun = spec_fun()
- Family1 = Family2 = family()

If Family1 is a family [page 263], then Family2 is the restriction [page 262] of Family1 to those elements i of the index set for which Fun applied to Family1[i] returns `true`. If Fun is a tuple {`external`,`Fun2`}, Fun2 is applied to the external set [page 264] of Family1[i], otherwise Fun is applied to Family1[i].

```
1> F1 = sofs:family([{a,[1,2,3]},{b,[1,2]},{c,[1]}]),
SpecFun = fun(S) -> sofs:no_elements(S) =:= 2 end,
F2 = sofs:family_specification(SpecFun, F1),
sofs:to_external(F2).
[{b,[1,2]}]
```

`family_to_digraph(Family [, GraphType]) -> Graph`

Types:

- Graph = digraph()
- Family = family()
- GraphType = -see digraph(3)-

Creates a directed graph from the family [page 263] Family. For each pair (a,{b[1],...,b[n]}) of Family, the vertex a as well the edges (a,b[i]) for 1<=i<=n are added to a newly created directed graph.

If no graph type is given, `digraph:new/1` is used for creating the directed graph, otherwise the GraphType argument is passed on as second argument to `digraph:new/2`.

It F is a family, it holds that F is a subset of `digraph_to_family(family_to_digraph(F),type(F))`. Equality holds if `union_of_family(F)` is a subset of `domain(F)`.

Creating a cycle in an acyclic graph exits the process with a `cyclic` message.

`family_to_relation(Family) -> BinRel`

Types:

- Family = family()
- BinRel = binary_relation()

If Family is a family [page 263], then BinRel is the binary relation containing all pairs (i,x) such that i belongs to the index set of Family and x belongs to Family[i].

```
1> F = sofs:family([{a,[]}, {b,[1]}, {c,[2,3]}]),
R = sofs:family_to_relation(F),
sofs:to_external(R).
[{b,1},{c,2},{c,3}]
```

`family_union(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 263] and Family1[i] is a set of sets for each i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the union [page 262] of Family1[i].

```
1> F1 = sofs:from_term([{a,[[1,2],[2,3]]},{b,[[]]}]),
F2 = sofs:family_union(F1),
sofs:to_external(F2).
[{a,[1,2,3]},{b,[]}]
```

`family_union(F)` is equivalent to `family_projection({sofs,union},F)`.

`family_union(Family1, Family2) -> Family3`

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 263], then Family3 is the family such that the index set is the union of Family1's and Family2's index sets, and Family3[i] is the union of Family1[i] and Family2[i] if both maps i, Family1[i] or Family2[i] otherwise.

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]},{c,[5,6]}]),
F2 = sofs:family([{b,[4,5]},{c,[7,8]},{d,[9,10]}]),
F3 = sofs:family_union(F1, F2),
sofs:to_external(F3).
[{a,[1,2]},{b,[3,4,5]},{c,[5,6,7,8]},{d,[9,10]}]
```

`field(BinRel) -> Set`

Types:

- BinRel = binary_relation()
- Set = set()

Returns the field [page 262] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:field(R),
sofs:to_external(S).
[1,2,a,b,c]
```

`field(R)` is equivalent to `union(domain(R), range(R))`.

`from_external(ExternalSet, Type) -> AnySet`

Types:

- ExternalSet = external_set()
- AnySet = anyset()
- Type = type()

Creates a set from the external set [page 264] ExternalSet and the type [page 264] Type. It is assumed that Type is a valid type [page 264] of ExternalSet.

`from_sets(ListOfSets) -> Set`

Types:

- Set = set()
- ListOfSets = [anyset()]

Returns the unordered set [page 264] containing the sets of the list ListOfSets.

```
1> S1 = sofs:relation([{a,1},{b,2}]),
S2 = sofs:relation([{x,3},{y,4}]),
S = sofs:from_sets([S1,S2]),
sofs:to_external(S).
[[{a,1},{b,2}],[{x,3},{y,4}]]
```

`from_sets(TupleOfSets) -> Ordset`

Types:

- Ordset = ordset()
- TupleOfSets = tuple-of(anyset())

Returns the ordered set [page 264] containing the sets of the non-empty tuple TupleOfSets.

`from_term(Term [, Type]) -> AnySet`

Types:

- AnySet = anyset()
- Term = term()
- Type = type()

Creates an element of Sets [page 264] by traversing the term Term, sorting lists, removing duplicates and deriving or verifying a valid type [page 264] for the so obtained external set. An explicitly given type [page 264] Type can be used to limit the depth of the traversal; an atomic type stops the traversal, as demonstrated by this example where "foo" and {"foo"} are left unmodified:

```
1> S = sofs:from_term([{{"foo"},[1,1]},{"foo",[2,2]}], [{atom,[atom]}]),
sofs:to_external(S).
[{{"foo"},[1]},{"foo",[2]}]
```

`from_term` can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets since all functions in this module that *do* anything operate on unordered sets. Creating unordered sets from a collection of ordered sets may be the way to go if the ordered sets are big and one does not want to waste heap by rebuilding the elements of the unordered set. An example showing that a set can be built "layer by layer":

```
1> A = sofs:from_term(a),
S = sofs:set([1,2,3]),
P1 = sofs:from_sets({A,S}),
P2 = sofs:from_term({b,[6,5,4]}),
Ss = sofs:from_sets([P1,P2]),
sofs:to_external(Ss).
[{a,[1,2,3]},{b,[4,5,6]}]
```

Other functions that create sets are `from_external/2` and `from_sets/1`. Special cases of `from_term/2` are `a_function/1,2`, `empty_set/0`, `family/1,2`, `relation/1,2`, and `set/1,2`.

`image(BinRel, Set1) -> Set2`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the image [page 262] of the set Set1 under the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([1,2]),
S2 = sofs:image(R, S1),
sofs:to_external(S2).
[a,b,c]
```

`intersection(SetOfSets) -> Set`

Types:

- Set = set()
- SetOfSets = set_of_sets()

Returns the intersection [page 262] of the set of sets SetOfSets.

Intersecting an empty set of sets exits the process with a `badarg` message.

`intersection(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the intersection [page 262] of Set1 and Set2.

`intersection_of_family(Family) -> Set`

Types:

- Family = family()
- Set = set()

Returns the intersection of the family [page 263] Family.

Intersecting an empty family exits the process with a `badarg` message.

```
1> F = sofs:family([{a,[0,2,4]},{b,[0,1,2]},{c,[2,3]}]),
S = sofs:intersection_of_family(F),
sofs:to_external(S).
[2]
```

`inverse(Function1) -> Function2`

Types:

- Function1 = Function2 = function()

Returns the inverse [page 263] of the function Function1.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
R2 = sofs:inverse(R1),
sofs:to_external(R2).
[{a,1},{b,2},{c,3}]
```

`inverse_image(BinRel, Set1) -> Set2`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the inverse image [page 262] of Set1 under the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([c,d,e]),
S2 = sofs:inverse_image(R, S1),
sofs:to_external(S2).
[2,3]
```

`is_a_function(BinRel) -> Bool`

Types:

- Bool = bool()
- BinRel = binary_relation()

Returns `true` if the binary relation BinRel is a function [page 263] or the untyped empty set, `false` otherwise.

`is_disjoint(Set1, Set2) -> Bool`

Types:

- Bool = bool()
- Set1 = Set2 = set()

Returns `true` if Set1 and Set2 are disjoint [page 262], `false` otherwise.

`is_empty_set(AnySet) -> Bool`

Types:

- AnySet = anyset()
- Bool = bool()

Returns `true` if Set is an empty unordered set, `false` otherwise.

`is_equal(AnySet1, AnySet2) -> Bool`

Types:

- AnySet1 = AnySet2 = anyset()
- Bool = bool()

Returns `true` if the AnySet1 and AnySet2 are equal [page 262], `false` otherwise.

`is_set(AnySet) -> Bool`

Types:

- AnySet = anyset()
- Bool = bool()

Returns `true` if AnySet is an unordered set [page 264], and `false` if AnySet is an ordered set or an atomic set.

`is_sofs_set(Term) -> Bool`

Types:

- Bool = bool()
- Term = term()

Returns `true` if Term is an unordered set [page 264], an ordered set or an atomic set, `false` otherwise.

`is_subset(Set1, Set2) -> Bool`

Types:

- Bool = bool()
- Set1 = Set2 = set()

Returns `true` if Set1 is a subset [page 262] of Set2, `false` otherwise.

`is_type(Term) -> Bool`

Types:

- Bool = bool()
- Term = term()

Returns `true` if the term Term is a type [page 264].

`join(Relation1, I, Relation2, J) -> Relation3`

Types:

- Relation1 = Relation2 = Relation3 = relation()
- I = J = integer() > 0

Returns the natural join [page 263] of the relations Relation1 and Relation2 on coordinates I and J.

```
1> R1 = sofs:relation([{a,x,1},{b,y,2}]),
R2 = sofs:relation([{1,f,g},{1,h,i},{2,3,4}]),
J = sofs:join(R1, 3, R2, 1),
sofs:to_external(J).
[{a,x,1,f,g},{a,x,1,h,i},{b,y,2,3,4}]
```

`multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2`

Types:

- TupleOfBinRels = tuple-of(BinRel)
- BinRel = BinRel1 = BinRel2 = binary_relation()

If TupleOfBinRels is a non-empty tuple {R[1],...,R[n]} of binary relations and BinRel1 is a binary relation, then BinRel2 is the multiple relative product [page 263] of the ordered set (R[i],...,R[n]) and BinRel1.

```
1> Ri = sofs:relation([{a,1},{b,2},{c,3}]),
R = sofs:relation([{a,b},{b,c},{c,a}]),
MP = sofs:multiple_relative_product({Ri, Ri}, R),
sofs:to_external(sofs:range(MP)).
[{1,2},{2,3},{3,1}]
```

`no_elements(ASet) -> NoElements`

Types:

- ASet = set() | ordset()
- NoElements = integer() >= 0

Returns the number of elements of the ordered or unordered set ASet.

`partition(SetOfSets) -> Partition`

Types:

- SetOfSets = set_of_sets()
- Partition = set()

Returns the partition [page 263] of the union of the set of sets SetOfSets such that two elements are considered equal if they belong to the same elements of SetOfSets.

```
1> Sets1 = sofs:from_term([[a,b,c],[d,e,f],[g,h,i]]),
Sets2 = sofs:from_term([[b,c,d],[e,f,g],[h,i,j]]),
P = sofs:partition(sofs:union(Sets1, Sets2)),
sofs:to_external(P).
[[a],[b,c],[d],[e,f],[g],[h,i],[j]]
```

`partition(SetFun, Set) -> Partition`

Types:

- SetFun = set_fun()
- Partition = set()
- Set = set()

Returns the partition [page 263] of Set such that two elements are considered equal if the results of applying SetFun are equal.

```
1> Ss = sofs:from_term([[a],[b],[c,d],[e,f]]),
SetFun = fun(S) -> sofs:from_term(sofs:no_elements(S)) end,
P = sofs:partition(SetFun, Ss),
sofs:to_external(P).
[[[a],[b]],[[c,d],[e,f]]]
```

`partition(SetFun, Set1, Set2) -> {Set3, Set4}`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = Set4 = set()

Returns a pair of sets that, regarded as constituting a set, forms a partition [page 263] of Set1. If the result of applying SetFun to an element of Set1 yields an element in Set2, the element belongs to Set3, otherwise the element belongs to Set4.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
{R2,R3} = sofs:partition(1, R1, S),
{sofs:to_external(R2),sofs:to_external(R3)}.
{[{2,b}],[{1,a},{3,c}]}
```

`partition(F,S1,S2)` is equivalent to `{restriction(F,S1,S2), drestriction(F,S1,S2)}`.

`partition_family(SetFun, Set) -> Family`

Types:

- Family = family()
- SetFun = set_fun()
- Set = set()

Returns the family [page 263] Family where the indexed set is a partition [page 263] of Set such that two elements are considered equal if the results of applying SetFun are the same value i. This i is the index that Family maps onto the equivalence class [page 263].

```
1> S = sofs:relation([{a,a,a,a},{a,a,b,b},{a,b,b,b}]),
SetFun = {external, fun({A,_,C,_}) -> {A,C} end},
F = sofs:partition_family(SetFun, S),
sofs:to_external(F).
[{{a,a},[{a,a,a,a}]},{{a,b},[{a,a,b,b},{a,b,b,b}]}]
```

`product(TupleOfSets) -> Relation`

Types:

- Relation = relation()
- TupleOfSets = tuple-of(set())

Returns the Cartesian product [page 263] of the non-empty tuple of sets TupleOfSets. If (x[1],...,x[n]) is an element of the n-ary relation Relation, then x[i] is drawn from element i of TupleOfSets.

```
1> S1 = sofs:set([a,b]),
S2 = sofs:set([1,2]),
S3 = sofs:set([x,y]),
P3 = sofs:product({S1,S2,S3}),
sofs:to_external(P3).
[{a,1,x},{a,1,y},{a,2,x},{a,2,y},{b,1,x},{b,1,y},{b,2,x},{b,2,y}]
```

`product(Set1, Set2) -> BinRel`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the Cartesian product [page 262] of Set1 and Set2.

```
1> S1 = sofs:set([1,2]),
S2 = sofs:set([a,b]),
R = sofs:product(S1, S2),
sofs:to_external(R).
[{1,a},{1,b},{2,a},{2,b}]
```

product(S1,S2) is equivalent to product({S1,S2}).

`projection(SetFun, Set1) -> Set2`

Types:

- SetFun = set_fun()
- Set1 = Set2 = set()

Returns the set created by substituting each element of Set1 by the result of applying SetFun to the element.

If SetFun is a number i>=1 and Set1 is a relation, then the returned set is the projection [page 263] of Set1 onto coordinate i.

```
1> S1 = sofs:from_term([{1,a},{2,b},{3,a}]),
S2 = sofs:projection(2, S1),
sofs:to_external(S2).
[a,b]
```

`range(BinRel) -> Set`

Types:

- BinRel = binary_relation()
- Set = set()

Returns the range [page 262] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:range(R),
sofs:to_external(S).
[a,b,c]
```

`relation(Tuples [, Type]) -> Relation`

Types:

- N = integer()
- Type = N | type()
- Relation = relation()
- Tuples = [tuple()]

Creates a relation [page 262]. relation(R,T) is equivalent to from_term(R,T), if T is a type [page 264] and the result is a relation. If Type is an integer N, then [{atom,...,atom}]), where the size of the tuple is N, is used as type of the relation. If no type is explicitly given, the size of the first tuple of Tuples is used if there is such a tuple. relation([]) is equivalent to relation([],2).

`relation_to_family(BinRel) -> Family`

Types:

- Family = family()
- BinRel = binary_relation()

Returns the family [page 263] Family such that the index set is equal to the domain [page 262] of the binary relation BinRel, and Family[i] is the image [page 262] of the set of i under BinRel.

```
1> R = sofs:relation([{b,1},{c,2},{c,3}]),
F = sofs:relation_to_family(R),
sofs:to_external(F).
[{b,[1]},{c,[2,3]}]
```

`relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2`

Types:

- TupleOfBinRels = tuple-of(BinRel)
- BinRel = BinRel1 = BinRel2 = binary_relation()

If TupleOfBinRels is a non-empty tuple {R[1],...,R[n]} of binary relations and BinRel1 is a binary relation, then BinRel2 is the relative product [page 263] of the ordered set (R[i],...,R[n]) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian product [page 263] of the ranges of R[i], rangeR[1]...rangeR[n], is used instead (intuitively, nothing is "lost").

```
1> TR = sofs:relation([{1,a},{1,aa},{2,b}]),
R1 = sofs:relation([{1,u},{2,v},{3,c}]),
R2 = sofs:relative_product({TR, R1}),
sofs:to_external(R2).
[{1,{a,u}},{1,{aa,u}},{2,{b,v}}]
```

Note that `relative_product({R1},R2)` is different from `relative_product(R1,R2)`; the tuple of one element is not identified with the element itself.

`relative_product(BinRel1, BinRel2) -> BinRel3`

Types:

- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 262] of the binary relations BinRel1 and BinRel2.

`relative_product1(BinRel1, BinRel2) -> BinRel3`

Types:

- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 262] of the converse [page 262] of the binary relation BinRel1 and the binary relation BinRel2.

```
1> R1 = sofs:relation([{1,a},{1,aa},{2,b}]),
R2 = sofs:relation([{1,u},{2,v},{3,c}]),
R3 = sofs:relative_product1(R1, R2),
sofs:to_external(R3).
[{a,u},{aa,u},{b,v}]
```

`relative_product1(R1,R2)` is equivalent to `relative_product(converse(R1),R2)`.

`restriction(BinRel1, Set) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the restriction [page 262] of the binary relation BinRel1 to Set.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([1,2,4]),
R2 = sofs:restriction(R1, S),
sofs:to_external(R2).
[{1,a},{2,b}]
```

`restriction(SetFun, Set1, Set2) -> Set3`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that yield an element in Set2 as the result of applying SetFun.

```
1> S1 = sofs:relation([{1,a},{2,b},{3,c}]),
S2 = sofs:set([b,c,d]),
S3 = sofs:restriction(2, S1, S2),
sofs:to_external(S3).
[{2,b},{3,c}]
```

`set(Terms [, Type]) -> Set`

Types:

- Set = set()
- Terms = [term()]
- Type = type()

Creates an unordered set [page 264]. `set(L,T)` is equivalent to `from_term(L,T)`, if the result is an unordered set. If no type [page 264] is explicitly given, `[atom]` is used as type of the set.

`specification(Fun, Set1) -> Set2`

Types:

- Fun = spec_fun()
- Set1 = Set2 = set()

Returns the set containing every element of Set1 for which Fun returns `true`. If Fun is a tuple `{external,Fun2}`, Fun2 is applied to the external set [page 264] of each element, otherwise Fun is applied to each element.

```
1> R1 = sofs:relation([{a,1},{b,2}]),
R2 = sofs:relation([{x,1},{x,2},{y,3}]),
S1 = sofs:from_sets([R1,R2]),
S2 = sofs:specification({sofs,is_a_function}, S1),
sofs:to_external(S2).
[[{a,1},{b,2}]]
```

`strict_relation(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns the strict relation [page 263] corresponding to the binary relation BinRel1.

```
1> R1 = sofs:relation([{1,1},{1,2},{2,1},{2,2}]),
R2 = sofs:strict_relation(R1),
sofs:to_external(R2).
[{1,2},{2,1}]
```

`substitution(SetFun, Set1) -> Set2`

Types:

- SetFun = set_fun()
- Set1 = Set2 = set()

Returns a function, the domain of which is Set1. The value of an element of the domain is the result of applying SetFun to the element.

```
1> L = [{a,1},{b,2}].
[{a,1},{b,2}]
2> sofs:to_external(sofs:projection(1,sofs:relation(L))).
[a,b]
3> sofs:to_external(sofs:substitution(1,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
4> SetFun = {external, fun({A,_}=E) -> {E,A} end},
sofs:to_external(sofs:projection(SetFun,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
```

The relation of equality between the elements of {a,b,c}:

```
1> I = sofs:substitution(fun(A) -> A end, sofs:set([a,b,c])),
sofs:to_external(I).
[{a,a},{b,b},{c,c}]
```

Let SetOfSets be a set of sets and BinRel a binary relation. The function that maps each element Set of SetOfSets onto the image [page 262] of Set under BinRel is returned by this function:

```
images(SetOfSets, BinRel) ->
   Fun = fun(Set) -> sofs:image(BinRel, Set) end,
   sofs:substitution(Fun, SetOfSets).
```

Here might be the place to reveal something that was more or less stated before, namely that external unordered sets are represented as sorted lists. As a consequence, creating the image of a set under a relation R may traverse all elements of R (to that comes the sorting of results, the image). In `images/2`, BinRel will be traversed once for each element of SetOfSets, which may take too long. The following efficient function could be used instead under the assumption that the image of each element of SetOfSets under BinRel is non-empty:

```
images2(SetOfSets, BinRel) ->
   CR = sofs:canonical_relation(SetOfSets),
   R = sofs:relative_product1(CR, BinRel),
   sofs:relation_to_family(R).
```

`symdiff(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the symmetric difference [page 262] (or the Boolean sum) of Set1 and Set2.

```
1> S1 = sofs:set([1,2,3]),
S2 = sofs:set([2,3,4]),
P = sofs:symdiff(S1, S2),
sofs:to_external(P).
[1,4]
```

`symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`

Types:

- Set1 = Set2 = Set3 = Set4 = Set5 = set()

Returns a triple of sets: Set3 contains the elements of Set1 that do not belong to Set2; Set4 contains the elements of Set1 that belong to Set2; Set5 contains the elements of Set2 that do not belong to Set1.

`to_external(AnySet) -> ExternalSet`

Types:

- ExternalSet = external_set()
- AnySet = anyset()

Returns the external set [page 264] of an atomic, ordered or unordered set.

`to_sets(ASet) -> Sets`

Types:

- ASet = set() | ordset()
- Sets = tuple_of(AnySet) | [AnySet]

Returns the elements of the ordered set ASet as a tuple of sets, and the elements of the unordered set ASet as a sorted list of sets without duplicates.

`type(AnySet) -> Type`

Types:

- AnySet = anyset()
- Type = type()

Returns the type [page 264] of an atomic, ordered or unordered set.

`union(SetOfSets) -> Set`

Types:

- Set = set()
- SetOfSets = set_of_sets()

Returns the union [page 262] of the set of sets SetOfSets.

`union(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the union [page 262] of Set1 and Set2.

`union_of_family(Family) -> Set`

Types:

- Family = family()
- Set = set()

Returns the union of the family [page 263] Family.

```
1> F = sofs:family([{a,[0,2,4]},{b,[0,1,2]},{c,[2,3]}]),
S = sofs:union_of_family(F),
sofs:to_external(S).
[0,1,2,3,4]
```

`weak_relation(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns a subset S of the weak relation [page 263] W corresponding to the binary relation BinRel1. Let F be the field [page 262] of BinRel1. The subset S is defined so that x S y if x W y for some x in F and for some y in F.

```
1> R1 = sofs:relation([{1,1},{1,2},{3,1}]),
R2 = sofs:weak_relation(R1),
sofs:to_external(R2).
[{1,1},{1,2},{2,2},{3,1},{3,3}]
```

See Also

dict(3) [page 77], digraph(3) [page 81], orddict(3) [page 221], ordsets(3) [page 222], sets(3) [page 247]

string

Erlang 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 occurrence 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, SeparatorList) -> Tokens`

Types:

- String = SeparatorList = string()
- Tokens = [string()]

Returns a list of tokens in String, separated by the characters in SeparatorList.

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(3) [page 242]). 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

Erlang Module

A behaviour module for implementing a supervisor, a process which supervises other processes called child processes. A child process can either be another supervisor or a worker process. Worker processes are normally implemented using one of the `gen_event`, `gen_fsm`, or `gen_server` behaviours. A supervisor implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build an hierarchical process structure called a supervision tree, a nice way to structure a fault tolerant application. Refer to *OTP Design Principles* for more information.

A supervisor assumes the definition of which child processes to supervise to be located in a callback module exporting a pre-defined set of functions.

Unless otherwise stated, all functions in this module will fail if the specified supervisor does not exist or if bad arguments are given.

Supervision Principles

The supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary.

The children of a supervisor is defined as a list of *child specifications*. When the supervisor is started, the child processes are started in order from left to right according to this list. When the supervisor terminates, it first terminates its child processes in reversed start order, from right to left.

A supervisor can have one of the following *restart strategies*:

- `one_for_one` - if one child process terminates and should be restarted, only that child process is affected.
- `one_for_all` - if one child process terminates and should be restarted, all other child processes are terminated and then all child processes are restarted.
- `rest_for_one` - if one child process terminates and should be restarted, the 'rest' of the child processes – i.e. the child processes after the terminated child process in the start order – are terminated. Then the terminated child process and all child processes after it are restarted.
- `simple_one_for_one` - a simplified `one_for_one` supervisor, where all child processes are dynamically added instances of the same process type, i.e. running the same code.

  The functions `terminate_child/2`, `delete_child/2` and `restart_child/2` are invalid for `simple_one_for_one` supervisors and will return `{error,simple_one_for_one}` if the specified supervisor uses this restart strategy.

To prevent a supervisor from getting into an infinite loop of child process terminations and restarts, a *maximum restart frequency* is defined using two integer values MaxR and MaxT. If more than MaxR restarts occur within MaxT seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

```
child_spec() = {Id,StartFunc,Restart,Shutdown,Type,Modules}
 Id = term()
 StartFunc = {M,F,A}
  M = F = atom()
  A = [term()]
 Restart = permanent | transient | temporary
 Shutdown = brutal_kill | int()>=0 | infinity
 Type = worker | supervisor
 Modules = [Module] | dynamic
  Module = atom()
```

- Id is a name that is used to identify the child specification internally by the supervisor.
- StartFunc defines the function call used to start the child process. It should be a module-function-arguments tuple {M,F,A} used as apply(M,F,A).

  The start function *must create and link to* the child process, and should return {ok,Child} or {ok,Child,Info} where Child is the pid of the child process and Info an arbitrary term which is ignored by the supervisor.

  The start function can also return ignore if the child process for some reason cannot be started, in which case the child specification will be kept by the supervisor but the non-existing child process will be ignored.

  If something goes wrong, the function may also return an error tuple {error,Error}.

  Note that the start_link functions of the different behaviour modules fulfill the above requirements.
- Restart defines when a terminated child process should be restarted. A permanent child process should always be restarted, a temporary child process should never be restarted and a transient child process should be restarted only if it terminates abnormally, i.e. with another exit reason than normal.
- Shutdown defines how a child process should be terminated. brutal_kill means the child process will be unconditionally terminated using exit(Child,kill). An integer timeout value means that the supervisor will tell the child process to terminate by calling exit(Child,shutdown) and then wait for an exit signal from the child process. If no exit signal is received within the specified time, the child process is unconditionally terminated using exit(Child,kill).

  If the child process is another supervisor, Shutdown should be set to infinity to give the subtree ample time to shutdown.
- Type specifies if the child process is a supervisor or a worker.

- `Modules` is used by the release handler during code replacement to determine which processes are using a certain module. As a rule of thumb `Modules` should be a list with one element `[Module]`, where `Module` is the name of the callback module, if the child process is a supervisor, gen_server or gen_fsm. If the child process is an event manager (gen_event) with a dynamic set of callback modules, `Modules` should be `dynamic`. See *SASL User's Guide* for more information.
- Internally, the supervisor also keeps track of the pid `Child` of the child process, or `undefined` if no pid exists.

Exports

`start_link(Module, Args) -> Result`
`start_link(SupName, Module, Args) -> Result`

Types:

- SupName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid}} | shutdown | term()

Creates a supervisor process, linked to the calling process, which calls `Module:init/1` to find out about restart strategy, maximum restart frequency and child processes. To ensure a synchronized start-up procedure, this function does not return until `Module:init/1` has returned and all child processes have been started.

If `SupName={local,Name}` the supervisor is registered locally as `Name` using `register/2`. If `SupName={global,Name}` the supervisor is registered globally as `Name` using `global:register_name/2`. If no name is provided, the supervisor is not registered. If there already exists a process with the specified `SupName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor and its child processes are successfully created (i.e. if all child process start functions return `{ok,Child}`, `{ok,Child,Info}`, or `ignore`) the function returns `{ok,Pid}`, where `Pid` is the pid of the supervisor.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor terminates with reason `normal`. If `Module:init/1` fails or returns an incorrect value, this function returns `{error,Term}` where `Term` is a term with information about the error, and the supervisor terminates with reason `Term`.

If any child process start function fails or returns an error tuple or an erroneous value, the function returns `{error,shutdown}` and the supervisor terminates all started child processes and then itself with reason `shutdown`.

`start_child(SupRef, ChildSpec) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- ChildSpec = child_spec() | [term()]
- Result = {ok,Child} | {ok,Child,Info} | {error,Error}
- Child = pid() | undefined
- Info = term()
- Error = already_present | {already_started,Child} | term()

Dynamically adds a child specification to the supervisor `SupRef` which starts the corresponding child process.

`SupRef` can be:

- the pid,
- `Name`, if the supervisor is locally registered,
- {`Name,Node`}, if the supervisor is locally registered at another node, or
- {`global,Name`}, if the supervisor is globally registered.

`ChildSpec` should be a valid child specification (unless the supervisor is a `simple_one_for_one` supervisor, see below). The child process will be started by using the start function as defined in the child specification.

If the case of a `simple_one_for_one` supervisor, the child specification defined in `Module:init/1` will be used and `ChildSpec` should instead be an arbitrary list of terms `List`. The child process will then be started by appending `List` to the existing start function arguments, i.e. by calling `apply(M, F, A++List)` where {`M,F,A`} is the start function defined in the child specification.

If there already exists a child specification with the specified `Id`, `ChildSpec` is discarded and the function returns {`error,already_present`} or {`error,`{`already_started,Child`}}, depending on if the corresponding child process is running or not.

If the child process start function returns {`ok,Child`} or {`ok,Child,Info`}, the child specification and pid is added to the supervisor and the function returns the same value.

If the child process start function returns `ignore`, the child specification is added to the supervisor, the pid is set to `undefined` and the function returns {`ok,undefined`}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the child specification is discarded and the function returns {`error,Error`} where `Error` is a term containing information about the error and child specification.

`terminate_child(SupRef, Id) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()
- Result = ok | {error,Error}
- Error = not_found | simple_one_for_one

Tells the supervisor `SupRef` to terminate the child process corresponding to the child specification identified by `Id`. The process, if there is one, is terminated but the child specification is kept by the supervisor. This means that the child process may be later be restarted by the supervisor. The child process can also be restarted explicitly by calling `restart_child/2`. Use `delete_child/2` to remove the child specification.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If there is no child specification with the specified `Id`, the function returns `{error,not_found}`.

`delete_child(SupRef, Id) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()
- Result = ok | {error,Error}
- Error = running | not_found | simple_one_for_one

Tells the supervisor `SupRef` to delete the child specification identified by `Id`. The corresponding child process must not be running, use `terminate_child/2` to terminate it.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If the child specification identified by `Id` exists but the corresponding child process is running, the function returns `{error,running}`. If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`.

`restart_child(SupRef, Id) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()
- Result = {ok,Child} | {ok,Child,Info} | {error,Error}
- Child = pid() | undefined
- Error = running | not_found | simple_one_for_one | term()

Tells the supervisor `SupRef` to restart a child process corresponding to the child specification identified by `Id`. The child specification must exist and the corresponding child process must not be running.

See `start_child/2` for a description of `SupRef`.

If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`. If the child specification exists but the corresponding process is already running, the function returns `{error,running}`.

If the child process start function returns `{ok,Child}` or `{ok,Child,Info}`, the pid is added to the supervisor and the function returns the same value.

If the child process start function returns `ignore`, the pid remains set to `undefined` and the function returns `{ok,undefined}`.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns {error,Error} where Error is a term containing information about the error.

`which_children(SupRef) -> [{Id,Child,Type,Modules}]`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term() | undefined
- Child = pid() | undefined
- Type = worker | supervisor
- Modules = [Module] | dynamic
- Module = atom()

Returns a list with information about all child specifications and child processes belonging to the supervisor SupRef.

See start_child/2 for a description of SupRef.

The information given for each child specification/process is:

- Id - as defined in the child specification or undefined in the case of a simple_one_for_one supervisor.
- Child - the pid of the corresponding child process, or undefined if there is no such process.
- Type - as defined in the child specification.
- Modules - as defined in the child specification.

`check_childspecs([ChildSpec]) -> Result`

Types:

- ChildSpec = child_spec()
- Result = ok | {error,Error}
- Error = term()

This function takes a list of child specification as argument and returns ok if all of them are syntactically correct, or {error,Error} otherwise.

CALLBACK FUNCTIONS

The following functions should be exported from a supervisor callback module.

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,{{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
- RestartStrategy = one_for_all | one_for_one | rest_for_one | simple_one_for_one
- MaxR = MaxT = int()>=0
- ChildSpec = child_spec()

Whenever a supervisor is started using `supervisor:start_link/2,3`, this function is called by the new process to find out about restart strategy, maximum restart frequency and child specifications.

`Args` is the `Args` argument provided to the start function.

`RestartStrategy` is the restart strategy and `MaxR` and `MaxT` defines the maximum restart frequency of the supervisor. `[ChildSpec]` is a list of valid child specifications defining which child processes the supervisor should start and monitor. See the discussion about Supervision Principles above.

Note that when the restart strategy is `simple_one_for_one`, the list of child specifications must be a list with one child specification only. (The `Id` is ignored). No child process is then started during the initialization phase, but all children are assumed to be started dynamically using `supervisor:start_child/2`.

The function may also return `ignore`.

SEE ALSO

gen_event(3) [page 154], gen_fsm(3) [page 163], gen_server(3) [page 173], sys(3) [page 300]

supervisor_bridge

Erlang Module

A behaviour module for implementing a supervisor_bridge, a process which connects a subsystem not designed according to the OTP design principles to a supervision tree. The supervisor_bridge sits between a supervisor and the subsystem. It behaves like a real supervisor to its own supervisor, but has a different interface than a real supervisor to the subsystem. Refer to *OTP Design Principles* for more information.

A supervisor_bridge assumes the functions for starting and stopping the subsystem to be located in a callback module exporting a pre-defined set of functions.

The `sys` module can be used for debugging a supervisor_bridge.

Unless otherwise stated, all functions in this module will fail if the specified supervisor_bridge does not exist or if bad arguments are given.

Exports

`start_link(Module, Args) -> Result`
`start_link(SupBridgeName, Module, Args) -> Result`

Types:

- SupBridgeName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a supervisor_bridge process, linked to the calling process, which calls `Module:init/1` to start the subsystem. To ensure a synchronized start-up procedure, this function does not return until `Module:init/1` has returned.

If `SupBridgeName={local,Name}` the supervisor_bridge is registered locally as `Name` using `register/2`. If `SupBridgeName={global,Name}` the supervisor_bridge is registered globally as `Name` using `global:register_name/2`. If no name is provided, the supervisor_bridge is not registered. If there already exists a process with the specified `SupBridgeName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor_bridge and the subsystem are successfully started the function returns `{ok,Pid}`, where `Pid` is is the pid of the supervisor_bridge.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor_bridge terminates with reason `normal`. If `Module:init/1` fails or returns an error tuple or an incorrect value, this function returns {`error,Term`} where `Term` is a term with information about the error, and the supervisor_bridge terminates with reason `Term`.

CALLBACK FUNCTIONS

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

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,Pid,State} | ignore | {error,Error}
- Pid = pid()
- State = term()
- Error = term()

Whenever a supervisor_bridge is started using `supervisor_bridge:start_link/2,3`, this function is called by the new process to start the subsystem and initialize.

`Args` is the `Args` argument provided to the start function.

The function should return {`ok,Pid,State`} where `Pid` is the pid of the main process in the subsystem and `State` is any term.

If later `Pid` terminates with a reason `Reason`, the supervisor bridge will terminate with reason `Reason` as well. If later the supervisor_bridge is stopped by its supervisor with reason `Reason`, it will call `Module:terminate(Reason,State)` to terminate.

If something goes wrong during the initialization the function should return {`error,Error`} where `Error` is any term, or `ignore`.

`Module:terminate(Reason, State)`

Types:

- Reason = shutdown | term()
- State = term()

This function is called by the supervisor_bridge when it is about to terminate. It should be the opposite of `Module:init/1` and stop the subsystem and do any necessary cleaning up. The return value is ignored.

`Reason` is `shutdown` if the supervisor_bridge is terminated by its supervisor. If the supervisor_bridge terminates because a a linked process (apart from the main process of the subsystem) has terminated with reason `Term`, `Reason` will be `Term`.

`State` is taken from the return value of `Module:init/1`.

SEE ALSO

supervisor(3) [page 290], sys(3) [page 300]

sys

Erlang 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, Module, OldVsn, Extra)`
`change_code(Name, Module, OldVsn, 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

Erlang 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 evaluations 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.

`now_diff(T2, T1) -> {Time, Value}`

Types:

- T1 = T2 = {MegaSecs, Secs, MicroSecs}
- MegaSecs = Secs = MicroSecs = integer()

Calculates the time difference `T2 - T1` in *microseconds*, where `T1` and `T2` probably are timestamp tuples returned from `erlang:now/0`.

`seconds(Seconds) -> Milliseconds`

Returns the number of milliseconds in `Seconds`.

`minutes(Minutes) -> Milliseconds`

Return 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 explicitly removed by a call to `cancel/1`.

win32reg

Erlang Module

win32reg provides read and write access to the registry on Windows. It is essentially a port driver wrapped around the Win32 API calls for accessing the registry.

The registry is a hierarchical database, used to store various system and software information in Windows. It is available in Windows 95 and Windows NT. It contains installation data, and is updated by installers and system programs. The Erlang installer updates the registry by adding data that Erlang needs.

The registry contains keys and values. Keys are like the directories in a file system, they form a hierarchy. Values are like files, they have a name and a value, and also a type.

Paths to keys are left to right, with sub-keys to the right and backslash between keys. (Remember that backslashes must be doubled in Erlang strings.) Case is preserved but not significant. Example:
`"\\hkey_local_machine\\software\\Ericsson\\Erlang\\5.0"` is the key for the installation data for the latest Erlang release.

There are six entry points in the Windows registry, top level keys. They can be abbreviated in the win32reg module as:

```
Abbrev.          Registry key
=======          ============
hkcr             HKEY_CLASSES_ROOT
current_user     HKEY_CURRENT_USER
hkcu             HKEY_CURRENT_USER
local_machine    HKEY_LOCAL_MACHINE
hklm             HKEY_LOCAL_MACHINE
users            HKEY_USERS
hku              HKEY_USERS
current_config   HKEY_CURRENT_CONFIG
hkcc             HKEY_CURRENT_CONFIG
dyn_data         HKEY_DYN_DATA
hkdd             HKEY_DYN_DATA
```

The key above could be written as `"\\hklm\\software\\ericsson\\erlang\\5.0"`.

The win32reg module uses a current key. It works much like the current directory. From the current key, values can be fetched, sub-keys can be listed, and so on.

Under a key, any number of named values can be stored. They have name, and types, and data.

Currently, the win32reg module supports storing only the following types: REG_DWORD, which is an integer, REG_SZ which is a string and REG_BINARY which is a binary. Other types can be read, and will be returned as binaries.

There is also a “default” value, which has the empty string as name. It is read and written with the atom `default` instead of the name.

Some registry values are stored as strings with references to environment variables, e.g. `"%SystemRoot%Windows"`. `SystemRoot` is an environment variable, and should be replaced with its value. A function `expand/1` is provided, so that environment variables surrounded in % can be expanded to their values.

For additional information on the Windows registry consult the Win32 Programmer's Reference.

Exports

`change_key(RegHandle, Key) -> ReturnValue`

Types:

- RegHandle = term()
- Key = string()

Changes the current key to another key. Works like cd. The key can be specified as a relative path or as an absolute path, starting with \.

`change_key_create(RegHandle, Key) -> ReturnValue`

Types:

- RegHandle = term()
- Key = string()

Creates a key, or just changes to it, if it is already there. Works like a combination of `mkdir` and `cd`. Calls the Win32 API function `RegCreateKeyEx()`.

The registry must have been opened in write-mode.

`close(RegHandle)-> ReturnValue`

Types:

- RegHandle = term()

Closes the registry. After that, the `RegHandle` cannot be used.

`current_key(RegHandle) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = {ok, string()}

Returns the path to the current key. This is the equivalent of `pwd`.

Note that the current key is stored in the driver, and might be invalid (e.g. if the key has been removed).

`delete_key(RegHandle) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = ok | {error, ErrorId}

Deletes the current key, if it is valid. Calls the Win32 API function `RegDeleteKey()`. Note that this call does not change the current key, (unlike `change_key_create/2`.) This means that after the call, the current key is invalid.

`delete_value(RegHandle, Name) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = ok | {error, ErrorId}

Deletes a named value on the current key. The atom `default` is used for the the default value.

The registry must have been opened in write-mode.

`expand(String) -> ExpandedString`

Types:

- String = string()
- ExpandedString = string()

Expands a string containing environment variables between percent characters. Anything between two % is taken for a environment variable, and is replaced by the value. Two consecutive % is replaced by one %.

A variable name that is not in the environment, will result in an error.

`format_error(ErrorId) -> ErrorString`

Types:

- ErrorId = atom()
- ErrorString = string()

Convert an POSIX errorcode to a string (by calling `erl_posix_msg:message`).

`open(OpenModeList)-> ReturnValue`

Types:

- OpenModeList = [OpenMode]
- OpenMode = read | write

Opens the registry for reading or writing. The current key will be the root (`HKEY_CLASSES_ROOT`). The `read` flag in the mode list can be omitted.

Use `change_key/2` with an absolute path after `open`.

`set_value(RegHandle, Name, Value) -> ReturnValue`

Types:

- Name = string() | default
- Value = string() | integer() | binary()

Sets the named (or default) value to value. Calls the Win32 API function `RegSetValueEx()`. The value can be of three types, and the corresponding registry type will be used. Currently the types supported are: `REG_DWORD` for integers, `REG_SZ` for strings and `REG_BINARY` for binaries. Other types cannot currently be added or changed.

The registry must have been opened in write-mode.

`sub_keys(RegHandle) -> ReturnValue`

Types:

- ReturnValue = {ok, SubKeys} | {error, ErrorId}
- SubKeys = [SubKey]
- SubKey = string()

Returns a list of subkeys to the current key. Calls the Win32 API function `EnumRegKeysEx()`.

Avoid calling this on the root keys, it can be slow.

`value(RegHandle, Name) -> ReturnValue`

Types:

- Name = string() | default
- ReturnValue = {ok, Value}
- Value = string() | integer() | binary()

Retrieves the named value (or default) on the current key. Registry values of type `REG_SZ`, are returned as strings. Type `REG_DWORD` values are returned as integers. All other types are returned as binaries.

`values(RegHandle) -> ReturnValue`

Types:

- ReturnValue = {ok, ValuePairs}
- ValuePairs = [ValuePair]
- ValuePair = {Name, Value}
- Name = string | default
- Value = string() | integer() | binary()

Retrieves a list of all values on the current key. The values have types corresponding to the registry types, see `value`. Calls the Win32 API function `EnumRegValuesEx()`.

SEE ALSO

Win32 Programmer's Reference (from Microsoft)

`erl_posix_msg`

The Windows 95 Registry (book from O'Reilly)

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.