STDLIB

version 1.12
Contents

1 STDLIB Reference Manual

1.1 beam_lib .............................................. 47
1.2 c ..................................................... 51
1.3 calendar ............................................... 55
1.4 dets .................................................. 61
1.5 dict .................................................. 76
1.6 digraph ............................................... 80
1.7 digraph_utils ...................................... 87
1.8 epp ................................................... 91
1.9 erl_eval ............................................ 93
1.10 erl_id_trans ...................................... 96
1.11 erl_internal ....................................... 97
1.12 erl_int ............................................. 99
1.13 erl_parse ........................................ 102
1.14 erl_pp ............................................. 105
1.15 erl_scan ........................................... 108
1.16 erl_tar ............................................ 110
1.17 ets .................................................. 116
1.18 file_sorter ......................................... 132
1.19 filelib ............................................. 137
1.20 filename ............................................ 140
1.21 gb_sets ............................................ 145
1.22 gb_trees ........................................... 149
1.23 gen_event .......................................... 152
1.24 gen_fsm ........................................... 161
1.25 gen_server ......................................... 171
1.26 io .................................................... 180
1.27 io_lib .............................................. 189
1.28 lib .................................................. 192
1.29 lists ............................................... 193
<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.30</td>
<td>log_mf.h</td>
<td>205</td>
</tr>
<tr>
<td>1.31</td>
<td>math</td>
<td>206</td>
</tr>
<tr>
<td>1.32</td>
<td>ms_transform</td>
<td>208</td>
</tr>
<tr>
<td>1.33</td>
<td>orddict</td>
<td>219</td>
</tr>
<tr>
<td>1.34</td>
<td>ordsets</td>
<td>220</td>
</tr>
<tr>
<td>1.35</td>
<td>pg</td>
<td>221</td>
</tr>
<tr>
<td>1.36</td>
<td>pool</td>
<td>222</td>
</tr>
<tr>
<td>1.37</td>
<td>proc_lib</td>
<td>224</td>
</tr>
<tr>
<td>1.38</td>
<td>proplists</td>
<td>228</td>
</tr>
<tr>
<td>1.39</td>
<td>queue</td>
<td>233</td>
</tr>
<tr>
<td>1.40</td>
<td>random</td>
<td>237</td>
</tr>
<tr>
<td>1.41</td>
<td>regexp</td>
<td>239</td>
</tr>
<tr>
<td>1.42</td>
<td>sets</td>
<td>244</td>
</tr>
<tr>
<td>1.43</td>
<td>shell</td>
<td>247</td>
</tr>
<tr>
<td>1.44</td>
<td>shell_default</td>
<td>255</td>
</tr>
<tr>
<td>1.45</td>
<td>slave</td>
<td>256</td>
</tr>
<tr>
<td>1.46</td>
<td>sofs</td>
<td>259</td>
</tr>
<tr>
<td>1.47</td>
<td>string</td>
<td>282</td>
</tr>
<tr>
<td>1.48</td>
<td>supervisor</td>
<td>287</td>
</tr>
<tr>
<td>1.49</td>
<td>supervisor_bridge</td>
<td>294</td>
</tr>
<tr>
<td>1.50</td>
<td>sys</td>
<td>297</td>
</tr>
<tr>
<td>1.51</td>
<td>timer</td>
<td>304</td>
</tr>
<tr>
<td>1.52</td>
<td>win32reg</td>
<td>308</td>
</tr>
</tbody>
</table>
STDLIB Reference Manual

Short Summaries

- Erlang Module `beamlib` [page 47] – An interface to the BEAM file format
- Erlang Module `c` [page 51] – Command Interface Module
- Erlang Module `calendar` [page 55] – Local and universal time, day-of-the-week, date and time conversions
- Erlang Module `dets` [page 61] – A Disk Based Term Storage
- Erlang Module `dict` [page 76] – Key-Value Dictionary
- Erlang Module `digraph` [page 80] – Directed Graphs
- Erlang Module `digraph-utils` [page 87] – Algorithms for Directed Graphs
- Erlang Module `epp` [page 91] – An Erlang Code Preprocessor
- Erlang Module `erl_eval` [page 93] – The Erlang Meta Interpreter
- Erlang Module `erl_id_trans` [page 96] – An Identity Parse Transform
- Erlang Module `erl_internal` [page 97] – Internal Erlang Definitions
- Erlang Module `erl_link` [page 99] – The Erlang Code Linter
- Erlang Module `erl_parse` [page 102] – The Erlang Parser
- Erlang Module `erl_pp` [page 105] – The Erlang Pretty Printer
- Erlang Module `erl_scan` [page 108] – The Erlang Token Scanner
- Erlang Module `erl_tar` [page 110] – Unix ‘tar’ utility for reading and writing tar archives
- Erlang Module `ets` [page 116] – Built-In Term Storage
- Erlang Module `file_sorter` [page 132] – File Sorter
- Erlang Module `filelib` [page 137] – File utilities, such as wildcard matching of filenames
- Erlang Module `filename` [page 140] – File Name Manipulation Functions
- Erlang Module `gb_sets` [page 145] – General Balanced Trees
- Erlang Module `gb_trees` [page 149] – General Balanced Trees
- Erlang Module `gen_event` [page 152] – Generic Event Handling Behaviour
- Erlang Module `gen_fsm` [page 161] – Generic Finite State Machine Behaviour
- Erlang Module `gen_server` [page 171] – Generic Server Behaviour
- Erlang Module `io` [page 180] – Standard I/O Server Interface Functions
- Erlang Module `io_lib` [page 189] – IO Library Functions
Erlang Module lib [page 192] – Interface Module
Erlang Module lists [page 193] – List Processing Functions
Erlang Module log.mfh [page 205] – An Event Handler which Logs Events to Disk
Erlang Module math [page 206] – Mathematical Functions
Erlang Module ms.transform [page 208] – Parse_transform that translates fun syntax into match specifications
Erlang Module orddict [page 219] – Key-Value Dictionary as Ordered List
Erlang Module ordssets [page 220] – Functions for Manipulating Sets as Ordered Lists
Erlang Module pg [page 221] – Distributed, Named Process Groups
Erlang Module pool [page 222] – Load Distribution Facility
Erlang Module proc.lib [page 224] – Plug-in Replacements for spawn/1,2,3,4, spawn_link/1,2,3,4, and spawn_opt/2,3,4,5.
Erlang Module proplists [page 228] – Support functions for property lists
Erlang Module queue [page 233] – Abstract Data Type for FIFO Queues
Erlang Module random [page 237] – Pseudo random number generation
Erlang Module regexp [page 239] – Regular Expression Functions for Strings
Erlang Module sets [page 244] – Functions for Set Manipulation
Erlang Module shell [page 247] – The Erlang Shell
Erlang Module shell.default [page 255] – Customizing the Erlang Environment
Erlang Module slave [page 256] – Functions to Starting and Controlling Slave Nodes
Erlang Module sofs [page 259] – Functions for Manipulating Sets of Sets
Erlang Module string [page 282] – String Processing Functions
Erlang Module sys [page 297] – A Functional Interface to System Messages
Erlang Module timer [page 304] – Timer Functions
Erlang Module win32reg [page 308] – win32reg provides access to the registry on Windows

beam_lib

The following functions are exported:
- chunks(FileNameOrBinary, [ChunkRef]) -> {ok, {Module, [ChunkData]}} | {error, Module, Reason} [page 48] Read selected chunks from a BEAM file or binary
- version(FileNameOrBinary) -> {ok, {Module, Version}} | {error, Module, Reason} [page 48] Read the BEAM file's module version.
• info(FileNameOrBinary) -> [SourceRef, {module, Module}, {chunks, [ChunkInfo]}] | {error, Module, Reason}  
  [page 48] Return some information about a BEAM file
• cmp(FileNameOrBinary, FileNameOrBinary) -> ok | {error, Module, Reason}  
  [page 48] Compare two BEAM files
• cmp dirs(Directory1, Directory2) -> {Only1, Only2, Different} | {error, Module, Reason}  
  [page 49] Compare the BEAM files in two directories
• diff dirs(Directory1, Directory2) -> ok | {error, Module, Reason}  
  [page 49] Compares the BEAM files in two directories
• strip(FileNameOrBinary) -> {ok, {Module, FileNameOrBinary}} | {error, Module, Reason}  
  [page 49] Removes chunks not needed by the loader from a BEAM file
• strip dirs(Files) -> {ok, [{Module, FileNameOrBinary}]} | {error, Module, Reason}  
  [page 49] Removes chunks not needed by the loader from BEAM files
• strip release(Directory) -> {ok, [{Module, FileName}]} | {error, Module, Reason}  
  [page 50] Removes chunks not needed by the loader from all BEAM files of a release
• format error(Error) -> character list()  
  [page 50] Return an English description of a BEAM read error reply

C

The following functions are exported:
• bt(Pid) -> void()  
  [page 51] Evaluate erlang:process_display(Pid, backtrace)
• c(File) -> CompileResult  
  [page 51] Compile a file
• c(File, Flags) -> CompileResult  
  [page 51] Compile a file
• cd(Dir) -> void()  
  [page 51] Change directory
• flush() -> void()  
  [page 52] Flush the shell message queue
• help() -> void()  
  [page 52] Display help information
• i() -> void()  
  [page 52] Display system information
• i(X, Y, Z) -> void()  
  [page 52] Evaluate process_info(pid(X, Y, Z))
• l(Module) -> void()  
  [page 52] Load code into the system
• lc(ListOfFile) -> Result  
  [page 52] Compile several files
• \text{ls()} \rightarrow \text{void()}
  \text{[page 52] List files}
• \text{ls(Dir)} \rightarrow \text{void()}
  \text{[page 52] List files in Dir}
• \text{m()} \rightarrow \text{void()}
  \text{[page 52] List all loaded modules}
• \text{m(Module)} \rightarrow \text{void()}
  \text{[page 53] Display information about a module}
• \text{memory()} \rightarrow \text{MemoryInformation}
  \text{[page 53] Returns memory information on dynamically allocated memory}
• \text{memory(MemoryTypeSpecification)} \rightarrow \text{MemoryInformation}
  \text{[page 53] Returns memory information on dynamically allocated memory}
• \text{nc(File)} \rightarrow \text{void()}
  \text{[page 53] Compile file and loads it on multiple nodes}
• \text{nc(File, Flags)} \rightarrow \text{void()}
  \text{[page 53] Compile file and loads it on multiples nodes}
• \text{ni()} \rightarrow \text{void()}
  \text{[page 53] Display network information}
• \text{nl(Module)} \rightarrow \text{void()}
  \text{[page 53] Load module in a network}
• \text{nregs()} \rightarrow \text{void()}
  \text{[page 53] Display registered processes on all nodes}
• \text{pid(X, Y, Z)} \rightarrow \text{pid()}
  \text{[page 53] Make a Pid}
• \text{pwd()} \rightarrow \text{void()}
  \text{[page 54] Print current working directory}
• \text{q()} \rightarrow \text{void()}
  \text{[page 54] Stop the Erlang node}
• \text{regs()} \rightarrow \text{void()}
  \text{[page 54] Display registered processes}
• \text{xm(ModSpec)} \rightarrow \text{void()}
  \text{[page 54] Cross reference check a module}
• \text{zi()} \rightarrow \text{void()}
  \text{[page 54] Display system information including zombies}

calendar

The following functions are exported:

• \text{date_to_gregorian_days(Year, Month, Day)} \rightarrow \text{Days}
  \text{[page 55] Compute the number of days from year 0 up to the given date.}
• \text{date_to_gregorian_days(Date)} \rightarrow \text{Days}
  \text{[page 55] Compute the number of days from year 0 up to the given date.}
• \text{datetime_to_gregorian_seconds(DateTime)} \rightarrow \text{Days}
  \text{[page 55] Compute the number of seconds from year 0 up to the given date and time.}
The following functions are exported:

- `all()` -> [Name]
  [page 62] Return a list of the names of all open Dets tables on this node.
- `bchunk(Name, Continuation) -> {Continuation2, Data} | 'end_of_table' | {error, Reason}
  [page 62] Return a chunk of objects stored in a Dets table.

- `close(Name) -> ok | {error, Reason}
  [page 63] Close a Dets table.

- `delete(Name, Key) -> ok | {error, Reason}
  [page 63] Delete all objects with a given key from a Dets table.

- `delete_objects(Name) -> ok | {error, Reason}
  [page 63] Delete all objects from a Dets table.

- `delete_object(Name, Object) -> ok | {error, Reason}
  [page 63] Delete a given object from a Dets table.

- `first(Name) -> Key | 'end_of_table'
  [page 63] Return the first key stored in a Dets table.

- `foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}
  [page 64] Fold a function over a Dets table.

- `foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}
  [page 64] Fold a function over a Dets table.

- `from_ets(Name, EtsTab) -> ok | {error, Reason}
  [page 64] Replace the objects of a Dets table with the objects of an Ets table.

- `info(Name) -> InfoList | undefined
  [page 64] Return information about a Dets table.

- `info(Name, Item) -> Value | undefined
  [page 65] Return the information associated with a given item for a Dets table.

- `init_table(Name, InitFun [, Options]) -> ok | {error, Reason}
  [page 65] Replace all objects of a Dets table.

- `insert(Name, Objects) -> ok | {error, Reason}
  [page 66] Insert one or more objects into a Dets table.

- `is_compatible_bchunk_format(Name, BchunkFormat) -> Bool
  [page 66] Test compatibility of a table's chunk data.

- `is_dets_file(FileName) -> Bool | {error, Reason}
  [page 67] Test for a Dets table.

- `lookup(Name, Key) -> [Object] | {error, Reason}
  [page 67] Return all objects with a given key stored in a Dets table.

- `match(Continuation) -> [{Match}, Continuation2] | 'end_of_table' | {error, Reason}
  [page 67] Match a chunk of objects stored in a Dets table and return a list of variable bindings.

- `match(Name, Pattern) -> [{Match}, Continuation] | 'end_of_table' | {error, Reason}
  [page 67] Match the objects stored in a Dets table and return a list of variable bindings.

- `match(Name, Pattern, N) -> [[Match], Continuation] | 'end_of_table' | {error, Reason}
  [page 68] Match the first chunk of objects stored in a Dets table and return a list of variable bindings.

- `match_delete(Name, Pattern) -> N | {error, Reason}
  [page 68] Delete all objects that match a given pattern from a Dets table.
The following functions are exported:

- `match_object(Continuation) -> [{Object}, Continuation2] | "$end_of_table" | {error, Reason}
  [page 68] Match a chunk of objects stored in a Dets table and return a list of objects.
- `match_object(Name, Pattern) -> [Object] | {error, Reason}
  [page 69] Match the objects stored in a Dets table and return a list of objects.
- `match_object(Name, Pattern, N) -> [{Object}, Continuation] | "$end_of_table" | {error, Reason}
  [page 69] Match the first chunk of objects stored in a Dets table and return a list of objects.
- `member(Name, Key) -> Bool | {error, Reason}
  [page 69] Test for occurrence of a key in a Dets table.
- `next(Name, Key1) -> Key2 | "$end_of_table"
  [page 70] Return the next key in a Dets table.
- `open_file(Filename) -> {ok, Reference} | {error, Reason}
  [page 70] Open an existing Dets table.
- `open_file(Name, Args) -> {ok, Name} | {error, Reason}
  [page 70] Open a Dets table.
- `pid2name(Pid) -> {ok, Name} | undefined
  [page 71] Return the name of the Dets table handled by a pid.
- `safe_fixtable(Name, Fix)
  [page 71] Fix a Dets table for safe traversal.
- `select(Continuation) -> {Selection, Continuation2} | "$end_of_table" | {error, Reason}
  [page 72] Apply a match specification to some objects stored in a Dets table.
- `select(Name, MatchSpec) -> Selection | {error, Reason}
  [page 72] Apply a match specification to all objects stored in a Dets table.
- `select(Name, MatchSpec, N) -> {Selection, Continuation} | "$end_of_table" | {error, Reason}
  [page 72] Apply a match specification to the first chunk of objects stored in a Dets table.
- `select_delete(Name, MatchSpec) -> N | {error, Reason}
  [page 73] Delete all objects that match a given pattern from a Dets table.
- `slot(Name, I) -> "$end_of_table" | [Object] | {error, Reason}
  [page 73] Return the list of objects associated with a slot of a Dets table.
- `sync(Name) -> ok | {error, Reason}
  [page 73] Ensure that all updates made to a Dets table are written to disk.
- `to_ets(Name, EtsTab) -> EtsTab | {error, Reason}
  [page 73] Insert all objects of a Dets table into an Ets table.
- `traverse(Name, Fun) -> Return | {error, Reason}
  [page 74] Apply a function to all or some objects stored in a Dets table.
- `update_counter(Name, Key, Increment) -> Result
  [page 74] Update a counter object stored in a Dets table.
- `append(Key, Value, Dict1) -> Dict2`  
  [page 76] Append a value to keys in a dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`  
  [page 76] Append new values to keys in a dictionary
- `erase(Key, Dict1) -> Dict2`  
  [page 76] Erase a key from a dictionary
- `fetch(Key, Dict) -> Value`  
  [page 76] Look-up values in a dictionary
- `fetch_keys(Dict) -> Keys`  
  [page 77] Return all keys in a dictionary
- `filter(Pred, Dict1) -> Dict2`  
  [page 77] Choose elements which satisfy a predicate
- `find(Key, Dict) -> Result`  
  [page 77] Search for a key in a dictionary
- `fold(Function, Acc0, Dict) -> Acc1`  
  [page 77] Fold a function over a dictionary
- `from_list(List) -> Dict`  
  [page 77] Convert a list of pairs to a dictionary
- `is_key(Key, Dict) -> bool()`  
  [page 77] Test if a key is in a dictionary.
- `map(Func, Dict1) -> Dict2`  
  [page 77] Map a function over a dictionary
- `merge(Func, Dict1, Dict2) -> Dict3`  
  [page 78] Merge two dictionaries
- `new() -> dictionary()`  
  [page 78] Create a dictionary
- `store(Key, Value, Dict1) -> Dict2`  
  [page 78] Store a value in a dictionary
- `to_list(Dict) -> List`  
  [page 78] Convert a dictionary to a list of pairs
- `update(Key, Function, Dict) -> Dict`  
  [page 78] Update a value in a dictionary
- `update(Key, Function, Initial, Dict) -> Dict`  
  [page 79] Update a value in a dictionary
- `update_counter(Key, Increment, Dict) -> Dict`  
  [page 79] Increment a value in a dictionary

**digraph**

The following functions are exported:

- `add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 80] Add an edge to a digraph.
- `add_edge(G, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 80] Add an edge to a digraph.
- `add_edge(G, V1, V2) -> edge() | {error, Reason}`  
  [page 80] Add an edge to a digraph.
• add_vertex(G, V, Label) -> vertex()
  [page 81] Add or modify a vertex of a digraph.
• add_vertex(G, V) -> vertex()
  [page 81] Add or modify a vertex of a digraph.
• add_vertex(G) -> vertex()
  [page 81] Add or modify a vertex of a digraph.
• del_edge(G, E) -> true
  [page 81] Delete an edge from a digraph.
• del_edges(G, Edges) -> true
  [page 81] Delete edges from a digraph.
• del_path(G, V1, V2) -> true
  [page 81] Delete paths from a digraph.
• del_vertex(G, V) -> true
  [page 82] Delete a vertex from a digraph.
• del_vertices(G, Vertices) -> true
  [page 82] Delete vertices from a digraph.
• delete(G) -> true
  [page 82] Delete a digraph.
• edge(G, E) -> {E, V1, V2, Label} | false
  [page 82] Return the vertices and the label of an edge of a digraph.
• edges(G) -> Edges
  [page 82] Return all edges of a digraph.
• edges(G, V) -> Edges
  [page 82] Return all edges emanating from or incident on a vertex of a digraph.
• get_cycle(G, V) -> Vertices | false
  [page 83] Find one cycle in a digraph.
• get_path(G, V1, V2) -> Vertices | false
  [page 83] Find one path in a digraph.
• get_short_cycle(G, V) -> Vertices | false
  [page 83] Find one short cycle in a digraph.
• get_short_path(G, V1, V2) -> Vertices | false
  [page 83] Find one short path in a digraph.
• in_degree(G, V) -> integer()
  [page 84] Return the in-degree of a vertex of a digraph.
• in_edges(G, V) -> Edges
  [page 84] Return all edges incident on a vertex of a digraph.
• in_neighbours(G, V) -> Vertices
  [page 84] Return all in-neighbours of a vertex of a digraph.
• info(G) -> InfoList
  [page 84] Return information about a digraph.
• new() -> digraph()
  [page 85] Return a protected empty digraph, where cycles are allowed.
• new(Type) -> digraph() | {error, Reason}
  [page 85] Create a new empty digraph.
• no_edges(G) -> integer() >= 0
  [page 85] Return the number of edges of the digraph.
STDLIB Reference Manual

- no_vertices(G) -> integer() >= 0  
  [page 85] Return the number of vertices of a digraph.
- out_degree(G, V) -> integer()  
  [page 85] Return the out-degree of a vertex of a digraph.
- out_edges(G, V) -> Edges  
  [page 85] Return all edges emanating from a vertex of a digraph.
- out_neighbours(G, V) -> Vertices  
  [page 85] Return all out-neighbours of a vertex of a digraph.
- vertex(G, V) -> {V, Label} | false  
  [page 86] Return the label of a vertex of a digraph.
- vertices(G) -> Vertices  
  [page 86] Return all vertices of a digraph.

digraph_utils

The following functions are exported:

- components(Digraph) -> [Component]  
  [page 88] Return the components of a digraph.
- condensation(Digraph) -> CondensedDigraph  
  [page 88] Return a condensed graph of a digraph.
- cyclic_strong_components(Digraph) -> [StrongComponent]  
  [page 88] Return the cyclic strong components of a digraph.
- is_acyclic(Digraph) -> bool()  
  [page 88] Check if a digraph is acyclic.
- loop_vertices(Digraph) -> Vertices  
  [page 88] Return the vertices of a digraph included in some loop.
- postorder(Digraph) -> Vertices  
  [page 89] Return the vertices of a digraph in post-order.
- preorder(Digraph) -> Vertices  
  [page 89] Return the vertices of a digraph in pre-order.
- reachable(Vertices, Digraph) -> Vertices  
  [page 89] Return the vertices reachable from some vertices of a digraph.
- reachable_neighbours(Vertices, Digraph) -> Vertices  
  [page 89] Return the neighbours reachable from some vertices of a digraph.
- reaching(Vertices, Digraph) -> Vertices  
  [page 89] Return the vertices that reach some vertices of a digraph.
- reaching_neighbours(Vertices, Digraph) -> Vertices  
  [page 89] Return the neighbours that reach some vertices of a digraph.
- strong_components(Digraph) -> [StrongComponent]  
  [page 90] Return the strong components of a digraph.
- subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}  
  [page 90] Return a subgraph of a digraph.
- topsort(Digraph) -> Vertices | false  
  [page 90] Return a topological sorting of the vertices of a digraph.
epp

The following functions are exported:

- `open(FileName, IncludePath) -> {ok, Epp} | {error, ErrorDescriptor}`
  [page 91] Open a file for preprocessing
- `open(FileName, IncludePath, PredefMacros) -> {ok, Epp} | {error, ErrorDescriptor}`
  [page 91] Open a file for preprocessing
- `close(Epp) -> ok`
  [page 91] Close the preprocessing of the file associated with Epp
- `parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`
  [page 91] Return the next Erlang form from the opened Erlang source file
- `parse_file(FileName, IncludePath, PredefMacro) -> {ok, [Form]} | {error, OpenError}`
  [page 91] Preprocess and parse an Erlang source file

erl_eval

The following functions are exported:

- `exprs(Expressions, Bindings) -> {value, Value, NewBindings}`
  [page 93] Evaluate expressions
- `exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 93] Evaluate expressions
- `exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 93] Evaluate expressions
- `expr(Expression, Bindings) -> {value, Value, NewBindings}`
  [page 93] Evaluate expression
- `expr(Expression, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 93] Evaluate expression
- `expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 93] Evaluate expression
- `expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
  [page 94] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 94] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 94] Evaluate a list of expressions
- `new_bindings() -> BindingStruct`
  [page 94] Return a bindings structure
• bindings(BindingStruct) -> Bindings
  [page 94] Return bindings
• binding(Name, BindingStruct) -> Binding
  [page 94] Return bindings
• add_binding(Name, Value, Bindings) -> BindingStruct
  [page 94] Add a binding
• del_binding(Name, Bindings) -> BindingStruct
  [page 94] Delete a binding

erl_id_trans

The following functions are exported:

• parse_transform(Forms, Options) -> Forms
  [page 96] Transform Erlang forms

erl_internal

The following functions are exported:

• bif(Name, Arity) -> bool()
  [page 97] Test for an Erlang BIF
• guard_bif(Name, Arity) -> bool()
  [page 97] Test for an Erlang BIF allowed in guards
• type_test(Name, Arity) -> bool()
  [page 97] Test for a valid type test
• arith_op(OpName, Arity) -> bool()
  [page 97] Test for an arithmetic operator
• bool_op(OpName, Arity) -> bool()
  [page 97] Test for a Boolean operator
• comp_op(OpName, Arity) -> bool()
  [page 98] Test for a comparison operator
• list_op(OpName, Arity) -> bool()
  [page 98] Test for a list operator
• send_op(OpName, Arity) -> bool()
  [page 98] Test for a send operator
• op_type(OpName, Arity) -> Type
  [page 98] Return operator type
The following functions are exported:

- `module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}`  
  [page 99] Check a module for errors
- `module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}`  
  [page 99] Check a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}`  
  [page 99] Check a module for errors
- `is_guard_test(Expr) -> bool()`  
  [page 100] Test for a guard test
- `format_error(ErrorDescriptor) -> string()`  
  [page 100] Format an error descriptor

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`  
  [page 102] Parse an Erlang form
- `parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`  
  [page 102] Parse Erlang expressions
- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`  
  [page 102] Parse an Erlang term
- `format_error(ErrorDescriptor) -> string()`  
  [page 103] Format an error descriptor
- `tokens(AbsTerm) -> Tokens`  
  [page 103] Generate a list of tokens for an expression
- `tokens(AbsTerm, MoreTokens) -> Tokens`  
  [page 103] Generate a list of tokens for an expression
- `normalise(AbsTerm) -> Data`  
  [page 103] Convert abstract form to an Erlang term
- `abstract(Data) -> AbsTerm`  
  [page 103] Convert an Erlang term into an abstract form

The following functions are exported:

- `form(Form) -> DeepCharList`  
  [page 105] Pretty print a form
- `form(Form, HookFunction) -> DeepCharList`  
  [page 105] Pretty print a form
- `attribute(Attribute) -> DeepCharList`  
  [page 105] Pretty print an attribute
### STDLIB Reference Manual

- **attribute(Attribute, HookFunction)** → DeepCharList
  
  [page 105] Pretty print an attribute

- **function(Function)** → DeepCharList
  
  [page 105] Pretty print a function

- **function(Function, HookFunction)** → DeepCharList
  
  [page 105] Pretty print a function

- **guard(Guard)** → DeepCharList
  
  [page 105] Pretty print a guard

- **guard(Guard, HookFunction)** → DeepCharList
  
  [page 105] Pretty print a guard

- **exprs(Expressions)** → DeepCharList
  
  [page 106] Pretty print Expressions

- **exprs(Expressions, HookFunction)** → DeepCharList
  
  [page 106] Pretty print Expressions

- **exprs(Expressions, Indent, HookFunction)** → DeepCharList
  
  [page 106] Pretty print Expressions

- **expr(Expression)** → DeepCharList
  
  [page 106] Pretty print one Expression

- **expr(Expression, HookFunction)** → DeepCharList
  
  [page 106] Pretty print one Expression

- **expr(Expression, Indent, HookFunction)** → DeepCharList
  
  [page 106] Pretty print one Expression

- **expr(Expression, Indent, Precedence, HookFunction)** → DeepCharList
  
  [page 106] Pretty print one Expression

### erl_scan

The following functions are exported:

- **string(CharList, StartLine)** → \{ok, Tokens, EndLine\} | Error
  
  [page 108] Scan a string and returns the Erlang tokens

- **string(CharList)** → \{ok, Tokens, EndLine\} | Error
  
  [page 108] Scan a string and returns the Erlang tokens

- **tokens(Continuation, CharList, StartLine)** → Return
  
  [page 108] Re-entrant scanner

- **reserved_word(Atom)** → bool()
  
  [page 109] Test for a reserved word

- **format_error(Descriptor)** → string()
  
  [page 109] Format an error descriptor

### erl_tar

The following functions are exported:

- **add(TarDescriptor, Filename, Options)** → RetValue
  
  [page 111] Add a file to an open tar file
add(TarDescriptor, Filename, NameInArchive, Options) -> RetValue
[page 111] Add a file to an open tar file

close(TarDescriptor)
[page 111] Close an open tar file

create(Name, FileList) -> RetValue
[page 111] Create a tar archive

create(Name, FileList, OptionList)
[page 112] Create a tar archive with options

extract(Name) -> RetValue
[page 112] Extract all files from a tar file

extract(Name, OptionList)
[page 112] Extract files from a tar file

format_error(Reason) -> string()
[page 113] Convert error term to a readable string

open(Name, OpenModeList) -> RetValue
[page 113] Open a tar file.

table(Name) -> RetValue
[page 114] Retrieve the name of all files in a tar file

table(Name, Options)
[page 114] Retrieve name and information of all files in a tar file

t(Name)
[page 114] Print the name of each file in a tar file

tt(Name)
[page 114] Print name and information for each file in a tar file

ets

The following functions are exported:

all() -> [Tab]
[page 117] Return a list of all ETS tables

delete(Tab) -> true
[page 117] Delete an entire ETS table.

delete(Tab, Key) -> true
[page 117] Delete all objects with a given key from an ETS table.

delete_all_objects(Tab) -> true
[page 117] Delete all objects in an ETS table.

delete_object(Tab, Object) -> true
[page 117] Deletes a specific from an ETS table.

file2tab(Filename) -> {ok,Tab} | {error,Reason}
[page 117] Read an ETS table from a file.

first(Tab) -> Key | '$end_of_table'
[page 117] Return the first key in an ETS table.

fixtable(Tab, true|false) -> true | false
[page 118] Fixe an ETS table for safe traversal (obsolete).

foldl(Function, Acc0, Tab) -> Acc1
[page 118] Fold a function over an ETS table
• foldr(Function, Acc0, Tab) -> Acc1
  [page 118] Fold a function over an ETS table
• from_dets(Tab, DetsTab) -> Tab
  [page 118] Fill an ETS table with the objects from a DETS table.
• fun2ms(LiteralFun) -> MatchSpec
  [page 119] Pseudo function that transforms fun syntax to matchspec.
• i() -> void()
  [page 120] Display information about all ETS tables on tty.
• i(Tab) -> void()
  [page 120] Browse an ETS table on tty.
• info(Tab) -> [{Item,Value}] | undefined
  [page 120] Return information about an ETS table.
• info(Tab, Item) -> Value | undefined
  [page 121] Return the information associated with given item for an ETS table.
• init_table(Name, InitFun) -> true
  [page 121] Replace all objects of an ETS table.
• insert(Tab, ObjectOrObjects) -> true
  [page 121] Insert an object into an ETS table.
• last(Tab) -> Key | '$end of table'
  [page 122] Return the last key in an ETS table of type ordered_set.
• lookup(Tab, Key) -> [Object]
  [page 122] Return all objects with a given key in an ETS table.
• lookup_element(Tab, Key, Pos) -> Elem
  [page 122] Return the Pos:th element of all objects with a given key in an ETS table.
• match(Tab, Pattern) -> [Match]
  [page 122] Match the objects in an ETS table against a pattern.
• match(Tab, Pattern, Limit) -> {{Match},Continuation} | '$end of table'
  [page 123] Match the objects in an ETS table against a pattern and returns part of the answers.
• match(Continuation) -> {{Match},Continuation} | '$end of table'
• match_delete(Tab, Pattern) -> true
  [page 124] Delete all objects which match a given pattern from an ETS table.
• match_object(Tab, Pattern) -> [Object]
  [page 124] Match the objects in an ETS table against a pattern.
• match_object(Tab, Pattern, Limit) -> {{Match},Continuation} | '$end of table'
  [page 124] Match the objects in an ETS table against a pattern and returns part of the answers.
• match_object(Continuation) -> {{Match},Continuation} | '$end of table'
• member(Tab, Key) -> true | false
new(Name, Options) -> tid()
    [page 125] Create a new ETS table.
next(Tab, Key1) -> Key2 | '$end_of_table'
    [page 125] Return the next key in an ETS table.
prev(Tab, Key1) -> Key2 | '$end_of_table'
    [page 126] Return the previous key in an ETS table of type ordered_set.
rename(Tab, Name) -> Name
    [page 126] Rename a named ETS table.
safe_fixtable(Tab, true|false) -> true | false
    [page 126] Fix an ETS table for safe traversal.
select(Tab, MatchSpec) -> [Object]
    [page 127] Match the objects in an ETS table against a match_spec.
select(Tab, MatchSpec, Limit) -> [{[Match],Continuation}] | '$end_of_table'
    [page 128] Match the objects in an ETS table against a match_spec and returns part of the answers.
select(Continuation) -> [{[Match],Continuation}] | '$end_of_table'
    [page 129] Continues matching objects in an ETS table.
select_delete(Tab, MatchSpec) -> NumDeleted
    [page 129] Match the objects in an ETS table against a match_spec and deletes objects where the match_spec returns 'true'
select_count(Tab, MatchSpec) -> NumMatched
    [page 129] Match the objects in an ETS table against a match_spec and returns the number of objects for which the match_spec returned 'true'
slot(Tab, I) -> [Object] | '$end_of_table'
    [page 130] Return all objects in a given slot of an ETS table.
tab2file(Tab, Filename) -> ok | {error,Reason}
    [page 130] Dump an ETS table to a file.
tab2list(Tab) -> [Object]
    [page 130] Return a list of all objects in an ETS table.
test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}
to_dets(Tab, DetsTab) -> Tab
    [page 131] Fill a DETS table with the objects from an ETS table.
update_counter(Tab, Key, [Pos,Incr,Threshold,SetValue]) -> Result
    [page 131] Update a counter object in an ETS table.
update_counter(Tab, Key, [Pos,Incr]) -> Result
    [page 131] Update a counter object in an ETS table.
update_counter(Tab, Key, Incr) -> Result
    [page 131] Update a counter object in an ETS table.

file_sorter

The following functions are exported:

sort(FileName) -> Reply
    [page 135] Sort terms on files.
• sort(Input, Output) \rightarrow \text{Reply}  
  [page 135] Sort terms on files.
• sort(Input, Output, Options) \rightarrow \text{Reply}  
  [page 135] Sort terms on files.
• keysort(KeyPos, File) \rightarrow \text{Reply}  
  [page 135] Sort terms on files by key.
• keysort(KeyPos, Input, Output) \rightarrow \text{Reply}  
  [page 135] Sort terms on files by key.
• keysort(KeyPos, Input, Output, Options) \rightarrow \text{Reply}  
  [page 135] Sort terms on files by key.
• merge(FileNames, Output) \rightarrow \text{Reply}  
  [page 135] Merge terms on files.
• merge(FileNames, Output, Options) \rightarrow \text{Reply}  
  [page 135] Merge terms on files.
• keymerge(KeyPos, FileNames, Output) \rightarrow \text{Reply}  
  [page 136] Merge terms on files by key.
• keymerge(KeyPos, FileNames, Output, Options) \rightarrow \text{Reply}  
  [page 136] Merge terms on files by key.
• check(File) \rightarrow \text{Reply}  
  [page 136] Check whether terms on files are sorted.
• check(FileNames, Options) \rightarrow \text{Reply}  
  [page 136] Check whether terms on files are sorted.
• keycheck(KeyPos, File) \rightarrow \text{CheckReply}  
  [page 136] Check whether terms on files are sorted by key.
• keycheck(KeyPos, FileNames, Options) \rightarrow \text{Reply}  
  [page 136] Check whether terms on files are sorted by key.

filelib

The following functions are exported:

• ensure_dir(Name) \rightarrow \text{true}  
  [page 137] Ensure that all parent directories needed to create Name exist.
• file_size(Filename) \rightarrow \text{integer()}  
  [page 137] Return the size in bytes of the file.
• fold_files(Dir, RegExp, Recursive, Fun, AccIn) \rightarrow \text{AccOut}  
  [page 137] Fold over all files matching a regular expression.
• is_dir(Name) \rightarrow \text{true | false}  
  [page 137] Test whether Name refer to a directory or not.
• is_file(Name) \rightarrow \text{true | false}  
  [page 137] Test whether Name refer to a file or directory.
• last_modified(Name) \rightarrow \{\{\text{Year,Month,Day}\},\{\text{Hour,Min,Sec}\}\}  
  [page 138] Return the local date and time when a file was last modified.
• wildcard(Wildcard) \rightarrow \text{list()}  
• wildcard(Wildcard, Cwd) \rightarrow \text{list()}  
filename

The following functions are exported:

- `absname(Filename) -> Absname`
  [page 140] Convert a relative Filename to an absolute name
- `absname(Filename, Directory) -> Absname`
  [page 140] Convert the relative Filename to an absolute name, based on Directory.
- `basename(Filename)`
  [page 141] Return the part of the Filename after the last directory separator
- `basename(Filename, Ext) -> string()`
  [page 141] Return the last component of Filename with Ext stripped
- `dirname(Filename) -> string()`
  [page 141] Return the directory part of a path name
- `extension(Filename) -> string() | []`
  [page 141] Return the file extension
- `join(Components) -> string()`
  [page 142] Join a list of file name Components with directory separators
- `join(Name1, Name2) -> string()`
  [page 142] Join two file name components with directory separators.
- `nativename(Path) -> string()`
  [page 142] Return the native form of a file Path
- `pathtype(Path) -> absolute | relative | volumerelative`
  [page 142] Return the type of a Path
- `rootname(Filename) -> string()`
  [page 143] Return all characters in Filename, except the extension.
- `rootname(Filename, Ext) -> string()`
  [page 143] Return all characters in Filename, except the extension.
- `split(Filename) -> Components`
  [page 143] Return a list whose elements are the file name components of Filename.
- `find_src(Module) -> {SourceFile, Options}`
  [page 143] Find the Filename and compilation options for a compiled Module.
- `find_src(Module, Rules) -> {SourceFile, Options}`
  [page 143] Find the Filename and compilation options for a compiled Module.

**gb_sets**

The following functions are exported:

- `empty()`
  [page 145] get empty set
- `is_empty(S)`
  [page 145] check if empty
- `size(S)`
  [page 145] get number of elements
- `singleton(X)`  
  **[page 145]** new set with one element
- `is_member(X, S)`  
  **[page 145]** check for member
- `insert(X, S)`  
  **[page 146]** insert new element
- `add(X, S)`  
  **[page 146]** add element
- `delete(X, S)`  
  **[page 146]** delete element
- `delete_any(X, T)`  
  **[page 146]** removes key if present
- `balance(S)`  
  **[page 146]** rebalance tree representation
- `union(S1, S2)`  
  **[page 146]** union of set
- `union(Ss)`  
  **[page 146]** union of list of sets
- `intersection(S1, S2)`  
  **[page 146]** intersection of sets
- `intersection(Ss)`  
  **[page 146]** intersection of list of sets
- `difference(S1, S2)`  
  **[page 146]** difference of sets
- `is_subset(S1, S2)`  
  **[page 147]** check for subset
- `to_list(S)`  
  **[page 147]** get list from set
- `from_list(List)`  
  **[page 147]** make set from list
- `from_ordset(L)`  
  **[page 147]** make set from ordset
- `smallest(S)`  
  **[page 147]** return smallest element
- `largest(S)`  
  **[page 147]** return largest element
- `take_smallest(S)`  
  **[page 147]** extract smallest element
- `take_largest(S)`  
  **[page 147]** extract largest element
- `iterator(S)`  
  **[page 147]** make iterator on set
- `next(T)`  
  **[page 147]** traverse with iterator
- `filter(P, S)`  
  **[page 147]** filter with predicate
- fold(F, A, S) [page 148] fold with fun
- is_set(S) [page 148] not recommended

**gb_trees**

The following functions are exported:

- empty() [page 149] returns empty tree
- is_empty(T) [page 149] true if tree is empty
- size(T) [page 149] number of nodes in tree
- lookup(X, T) [page 149] looks up key in tree
- get(X, T) [page 149] retrieves value stored with key
- insert(X, V, T) [page 150] inserts key and value in tree
- update(X, V, T) [page 150] updates key to new value
- enter(X, V, T) [page 150] inserts or updates key with value
- delete(X, T) [page 150] removes key
- delete_any(X, T) [page 150] removes key if present
- balance(T) [page 150] rebalance tree
- is_defined(X, T) [page 150] check if key exist
- keys(T) [page 150] keys as list
- values(T) [page 150] values as list
- to_list(T) [page 150] keys and values as tuple-list
- from_orddict(L) [page 150] make tree from orddict
- smallest(T) [page 151] return smallest key and value
- largest(T) [page 151] return smallest key and value
- take_smallest(T) [page 151] extract smallest key and value
- `take_largest(T)`
  [page 151] extract smallest key and value
- `iterator(T)`
  [page 151] get iterator on tree
- `next(S)`
  [page 151] iterate using iterator

**gen_event**

The following functions are exported:

- `start() -> Result`
  [page 153] Create a generic event manager.
- `start(EventMgrName) -> Result`
  [page 153] Create a generic event manager.
- `start_link() -> Result`
  [page 153] Create a generic event manager.
- `start_link(EventMgrName) -> Result`
  [page 153] Create a generic event manager.
- `add_handler(EventMgrRef, Handler, Args) -> Result`
  [page 153] Add an event handler to a generic event manager.
- `add_sup_handler(EventMgrRef, Handler, Args) -> Result`
  [page 154] Add a supervised event handler to a generic event manager.
- `notify(EventMgrRef, Event) -> ok`
  [page 154] Notify an event manager about an event.
- `sync_notify(EventMgrRef, Event) -> ok`
  [page 154] Notify an event manager about an event.
- `call(EventMgrRef, Handler, Request) -> Result`
  [page 155] Make a synchronous call to a generic event manager.
- `call(EventMgrRef, Handler, Request, Timeout) -> Result`
  [page 155] Make a synchronous call to a generic event manager.
- `delete_handler(EventMgrRef, Handler, Args) -> Result`
  [page 155] Delete an event handler from a generic event manager.
- `swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 156] Replace an event handler in a generic event manager.
- `swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 157] Replace an event handler in a generic event manager.
- `which_handlers(EventMgrRef) -> [Handler]`
  [page 157] Return all event handlers installed in a generic event manager.
- `stop(EventMgrRef) -> ok`
  [page 157] Terminate a generic event manager.
- `Module:init(InitArgs) -> {ok,State}`
  [page 158] Initialize an event handler.
- `Module:handle_event(Event, State) -> Result`
  [page 158] Handle an event.
- Module:handle_call(Request, State) -> Result
  [page 159] Handle a synchronous request.
- Module:handle_info(Info, State) -> Result
  [page 159] Handle an incoming message.
- Module:terminate(Arg, State) -> term()
  [page 159] Clean up before deletion.
- Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
  [page 160] Update the internal state due to code replacement:

\[
\text{gen fsm}
\]

The following functions are exported:

- start(Module, Args, Options) -> Result
  [page 162] Create a generic FSM process.
- start(FsmName, Module, Args, Options) -> Result
  [page 162] Create a generic FSM process.
- start_link(Module, Args, Options) -> Result
  [page 162] Create a generic FSM process.
- start_link(FsmName, Module, Args, Options) -> Result
  [page 162] Create a generic FSM process.
- send_event(FsmRef, Event) -> ok
  [page 163] Send an event asynchronously to a generic FSM.
- send_all_state_event(FsmRef, Event) -> ok
  [page 163] Send an event asynchronously to a generic FSM.
- sync_send_event(FsmRef, Event) -> Reply
  [page 163] Send an event synchronously to a generic FSM.
- sync_send_event(FsmRef, Event, Timeout) -> Reply
  [page 163] Send an event synchronously to a generic FSM.
- sync_send_all_state_event(FsmRef, Event) -> Reply
  [page 164] Send an event synchronously to a generic FSM.
- sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply
  [page 164] Send an event synchronously to a generic FSM.
- reply(Caller, Reply) -> true
  [page 164] Send a reply to a caller.
- send_event_after(Time, Event) -> Ref
  [page 164] Send a delayed event internally in a generic FSM.
- start_timer(Time, Msg) -> Ref
  [page 165] Send a timeout event internally in a generic FSM.
- cancel_timer(Ref) -> RemainingTime | false
  [page 165] Cancel an internal timer in a generic FSM.
- Module:init(Args) -> Result
  [page 166] Initialize process and internal state name and state data.
- Module:StateMachine(Event, StateData) -> Result
  [page 166] Handle an asynchronous event.
- Module:handle_event(Event, StateName, StateData) -> Result
  [page 167] Handle an asynchronous event.
- Module:StateName(Event, From, StateData) -> Result
  [page 167] Handle a synchronous event.
- Module:handle_sync_event(Event, From, StateName, StateData) -> Result
  [page 168] Handle a synchronous event.
- Module:handle_info(Info, StateName, StateData) -> Result
  [page 168] Handle an incoming message.
- Module:terminate(Reason, StateName, StateData)
  [page 169] Clean up before termination.
- Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}
  [page 169] Update the state data due to code replacement.

**gen_server**

The following functions are exported:

- `start(Module, Args, Options) -> Result`
  [page 171] Create a generic server process.
- `start(ServerName, Module, Args, Options) -> Result`
  [page 171] Create a generic server process.
- `start_link(Module, Args, Options) -> Result`
  [page 171] Create a generic server process.
- `start_link(ServerName, Module, Args, Options) -> Result`
  [page 171] Create a generic server process.
- `call(ServerRef, Request) -> Reply`
  [page 172] Make a synchronous call to a generic server.
- `call(ServerRef, Request, Timeout) -> Reply`
  [page 172] Make a synchronous call to a generic server.
- `multi_call(Name, Request) -> Result`
  [page 173] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request) -> Result`
  [page 173] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request, Timeout) -> Result`
  [page 173] Make a synchronous call to several generic servers.
- `cast(ServerRef, Request) -> ok`
  [page 174] Send an asynchronous request to a generic server.
- `abcast(Name, Request) -> abcast`
  [page 174] Send an asynchronous request to several generic servers.
- `abcast(Nodes, Name, Request) -> abcast`
  [page 174] Send an asynchronous request to several generic servers.
- `reply(Client, Reply) -> true`
  [page 175] Send a reply to a client.
- `enter_loop(Module, Options, State, ServerName, Timeout) -> Result`
  [page 175] Enters the gen_server receive loop.
- `enter_loop(Module, Options, State, Timeout) -> Result`
  [page 175] Enters the gen_server receive loop.
• enter_loop(Module, Options, State, ServerName) -> _
  [page 175] Enters the gen_server receive loop.
• Module:init(Args) -> Result
  [page 176] Initialize process and internal state.
• Module:handle_call(Request, From, State) -> Result
  [page 176] Handle a synchronous request.
• Module:handle_cast(Request, State) -> Result
  [page 177] Handle an asynchronous request.
• Module:handle_info(Info, State) -> Result
  [page 177] Handle an incoming message.
• Module:terminate(Reason, State)
  [page 178] Clean up before termination.
• Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
  [page 178] Update the internal state due to code replacement.

io

The following functions are exported:
• put_chars([IoDevice,] Chars)
  [page 180] Write characters to standard output
• nl([IoDevice])
  [page 180] Output a newline
• get_chars([IoDevice,] Prompt, Count)
  [page 180] Read characters from standard input
• get_line([IoDevice,] Prompt)
  [page 180] Read a line from standard input
• setopts([IoDevice,] OptList)
  [page 180] Set options for standard input/output
• write([IoDevice,] Term)
  [page 181] Write a term
• read([IoDevice,] Prompt)
  [page 181] Read a term
• fwrite(Format)
  [page 181] Write formatted output
• format(Format)
  [page 181] Write formatted output
• fwrite([IoDevice,] Format, Arguments)
  [page 181] Write formatted output
• format([IoDevice,] Format, Arguments)
  [page 181] Write formatted output
• fread([IoDevice,] Prompt, Format)
  [page 185] Read formatted input
• scan_erl_exprs(Prompt)
  [page 186] Read Erlang tokens
• scan_erl_exprs([IoDevice,] Prompt, StartLine)
  [page 186] Read Erlang tokens
• `scan_erl_form(Prompt)`
  [page 186] Read Erlang tokens
• `scan_erl_form(IoDevice, Prompt[, StartLine])`
  [page 186] Read Erlang tokens
• `parse_erl_exprs(Prompt)`
  [page 187] Read Erlang expressions
• `parse_erl_exprs(IoDevice, Prompt[, StartLine])`
  [page 187] Read Erlang expressions
• `parse_erl_form(Prompt)`
  [page 187] Read Erlang form
• `parse_erl_form(IoDevice, Prompt[, StartLine])`
  [page 187] Read Erlang form

io_lib

The following functions are exported:

• `nl()`
  [page 189] Return a newline
• `write(Term)`
  [page 189] Write a term
• `write(Term, Depth)`
  [page 189] Write a term
• `print(Term)`
  [page 189] Pretty print a term
• `print(Term, Column, LineLength, Depth)`
  [page 189] Pretty print a term
• `fwrite(Format, Data)`
  [page 189] List formatted output
• `format(Format, Data)`
  [page 189] List formatted output
• `fread(Format, String)`
  [page 189] List formatted input
• `fread(Continuation, CharList, Format)`
  [page 190] Re-entrant formatted reader
• `write_atom(Atom)`
  [page 190] Return an atom
• `write_string(String)`
  [page 190] Return a string
• `write_char(Integer)`
  [page 190] Return a character
• `indentation(String, StartIndent)`
  [page 190] Indentation after printing string
• `char_list(CharList) -> bool()`
  [page 190] Test for a list of characters
• `deep_char_list(CharList)`
  [page 191] Test for a deep list of characters
• `printable_list(CharList)`
  [page 191] Test for a list of printable characters
The following functions are exported:

- `flush_receive() -> void()`
  [page 192] Flush messages
- `error_message(Format, Args)`
  [page 192] Print error message
- `programe() -> atom()`
  [page 192] Return Erlang starter
- `nonl(List1)`
  [page 192] Remove last newline
- `send(To, Msg)`
  [page 192] Send a message
- `sendw(To, Msg)`
  [page 192] Send a message and waits for an answer

The following functions are exported:

- `append(ListOfLists) -> List1`
  [page 193] Append a list of lists
- `append(List1, List2) -> List3`
  [page 193] Append two lists
- `concat(Things) -> string()`
  [page 194] Concatenate a list of atoms
- `delete(Element, List1) -> List2`
  [page 194] Delete an element in a list
- `duplicate(N, Element) -> List`
  [page 194] Make N copies of element
- `flatlength(DeepList) -> int()`
  [page 194] Length of flattened deep list
- `flatten(DeepList) -> List`
  [page 194] Flatten a deep list
- `flatten(DeepList, Tail) -> List`
  [page 194] Flatten a deep list
- `keydelete(Key, N, TupleList1) -> TupleList2`
  [page 195] Delete a tuple for a tuple list
- `keymember(Key, N, TupleList) -> bool()`
  [page 195] Test for a key in a list of tuples
- `keymerge(N, List1, List2)`
  [page 195] Merge two key-sorted lists
- `keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2`
  [page 195] Replace tuple in tuple list
- `keysearch(Key, N, TupleList) -> Result`
  [page 195] Extract value of key in a list of tuples
• keysort(N, List1) -> List2  
  [page 195] Sort a list by key

• last(List) -> Element  
  [page 196] Return last element in a list

• max(List) -> Max  
  [page 196] Return maximum element of list

• member(Element, List) -> bool()  
  [page 196] Test for membership of a list

• merge(ListOfLists) -> List1  
  [page 196] Merge a list of sorted lists

• merge(List1, List2) -> List3  
  [page 196] Merge two sorted lists

• merge(Fun, List1, List2) -> List  
  [page 196] Merge two sorted list

• merge3(List1, List2, List3) -> List4  
  [page 197] Merge three sorted lists

• min(List) -> Min  
  [page 197] Return minimum element of list

• nth(N, List) -> Element  
  [page 197] Extract element from a list

• nthtail(N, List1) -> List2  
  [page 197] Return the N’th tail in List1

• prefix(List1, List2) -> bool()  
  [page 197] Test for list prefix

• reverse(List1) -> List2  
  [page 197] Reverse a list

• reverse(List1, List2) -> List3  
  [page 198] Reverse a list appending a tail

• seq(From, To) -> [int()]  
  [page 198] Generate a sequence of integers

• seq(From, To, Incr) -> [int()]  
  [page 198] Generate a sequence of integers

• sort(List1) -> List2  
  [page 198] Sort a list

• sort(Fun, List1) -> List2  
  [page 198] Sort a list

• sublist(List, N) -> List1  
  [page 198] Return the first N elements of List

• sublist(List1, Start, Length) -> List2  
  [page 199] Return a sub-list of list

• subtract(List1, List2) -> List3  
  [page 199] Subtract the element in one list from another list

• suffix(List1, List2) -> bool()  
  [page 199] Test for list suffix

• sum(List) -> number()  
  [page 199] Return sum of elements in a list
- `ukeymerge(N, List1, List2)`  
  [page 199] Merge two key-sorted lists and remove consecutive duplicates
- `ukeysort(N, List1)`  
  [page 199] Sort a list by key and remove consecutive duplicates
- `umerge(ListOfLists) -> List1`  
  [page 200] Merge a list of sorted lists without duplicates
- `umerge(List1, List2) -> List3`  
  [page 200] Merge two sorted lists without duplicates
- `umerge(Fun, List1, List2) -> List`  
  [page 200] Merge two sorted lists without duplicates
- `umerge3(List1, List2, List3) -> List4`  
  [page 200] Merge three sorted lists without duplicates
- `usort(List1) -> List2`  
  [page 200] Sort a list and remove duplicates
- `usort(Fun, List1) -> List2`  
  [page 200] Sort a list and remove duplicates
- `all(Pred, List) -> bool()`  
  [page 201] Return true if all elements in the list satisfy `Pred`
- `any(Pred, List) -> bool()`  
  [page 201] Return true if any of the elements `X` in the list satisfies `Pred(X)`
- `dropwhile(Pred, List1) -> List2`  
  [page 201] Drop elements from `List1` while `Pred` is true
- `filter(Pred, List1) -> List2`  
  [page 201] Choose elements which satisfy a predicate
- `flatmap(Function, List1) -> Element`  
  [page 201] Map and flatten in one pass
- `foldl(Function, Acc0, List) -> Acc1`  
  [page 201] Fold a function over a list
- `foldr(Function, Acc0, List) -> Acc1`  
  [page 202] Fold a function over a list
- `foreach(Function, List) -> void()`  
  [page 202] Apply function to each element of a list
- `map(Func, List1) -> List2`  
  [page 202] Map a function over a list
- `mapfoldl(Function, Acc0, List1) -> {List2, Acc}`  
  [page 202] Map and fold in one pass
- `mapfoldr(Function, Acc0, List1) -> {List2, Acc}`  
  [page 203] Map and fold in one pass
- `split(N, List) -> {List1, List2}`  
  [page 203] Partition `List` into one list of length `N` and one with the rest of the elements
- `splitwith(Pred, List) -> {List1, List2}`  
  [page 203] Partition `List1` into two lists according to `Pred`
- `takewhile(Pred, List1) -> List2`  
  [page 203] Take elements from `List1` while `Pred` is true
log_mf.h

The following functions are exported:

- `init(Dir, MaxBytes, MaxFiles)`
  [page 205] Initiate the event handler
- `init(Dir, MaxBytes, MaxFiles, Pred) -> Args`
  [page 205] Initiate the event handler

math

The following functions are exported:

- `pi() -> float()`
  [page 206] A useful number
- `sin(X)`
  [page 206] Diverse math functions
- `cos(X)`
  [page 206] Diverse math functions
- `tan(X)`
  [page 206] Diverse math functions
- `asin(X)`
  [page 206] Diverse math functions
- `acos(X)`
  [page 206] Diverse math functions
- `atan(X)`
  [page 206] Diverse math functions
- `atan2(Y, X)`
  [page 206] Diverse math functions
- `sinh(X)`
  [page 206] Diverse math functions
- `cosh(X)`
  [page 206] Diverse math functions
- `tanh(X)`
  [page 206] Diverse math functions
- `asinh(X)`
  [page 206] Diverse math functions
- `acosh(X)`
  [page 206] Diverse math functions
- `atanh(X)`
  [page 206] Diverse math functions
- `exp(X)`
  [page 206] Diverse math functions
- `log(X)`
  [page 206] Diverse math functions
- `log10(X)`
  [page 206] Diverse math functions
- pow(X, Y)  
  [page 206] Diverse math functions
- sqrt(X)  
  [page 206] Diverse math functions
- erf(X) -> float()  
  [page 206] Error function.
- erfc(X) -> float()  
  [page 207] Another error function

**ms_transform**

The following functions are exported:

- parse_transform(Forms, Options) -> Forms  
  [page 217] Transforms Erlang abstract format containing calls to ets/dbg:fun2ms into literal match specifications.
- transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()  
  [page 217] Used when transforming fun's created in the shell into match specifications.
- format_error(Errcode) -> ErrorMessage  
  [page 218] Error formatting function as required by the parse_transform interface.

**orddict**

No functions are exported.

**ordsets**

No functions are exported.

**pg**

The following functions are exported:

- create(PgName)  
  [page 221] Create an empty group
- create(PgName, Node)  
  [page 221] Create an empty group on a node
- join(PgName, Pid)  
  [page 221] Join a Pid to a process group
- send(Pgname, Message)  
  [page 221] Send a message tuple to all members of a process group
- esend(PgName, Mess)  
  [page 221] Send a message tuple to all members of a process group except the current node
- members(PgName)  
  [page 221] >Return a list of the current members in the process group
pool

The following functions are exported:

- `start(Name)`
  [page 222] Start a new pool
- `start(Name, Args)`
  [page 222] Start a new pool
- `attach(Node)`
  [page 222] Ensure that a pool master is running
- `stop()`
  [page 222] Stop the pool and kill all the slave nodes
- `get_nodes()`
  [page 223] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args)`
  [page 223] Spawn a process on the expected lowest future loaded pool node
- `pspawn_link(Mod, Fun, Args)`
  [page 223] Spawn links a process on the expected lowest future loaded pool node
- `get_node()`
  [page 223] Return the node ID of the expected lowest future loaded node
- `new_node(Host, Name)`
  [page 223] Start a new node and attach it to an already existing pool

proc_lib

The following functions are exported:

- `spawn(Fun) -> Pid`
  [page 224] Spawn a new process.
- `spawn(Node, Fun) -> Pid`
  [page 224] Spawn a new process.
- `spawn(Module, Func, Args) -> Pid`
  [page 224] Spawn a new process.
- `spawn(Node, Module, Func, Args) -> Pid`
  [page 224] Spawn a new process.
- `spawn_link(Fun) -> Pid`
  [page 224] Spawn a new process and set a link.
- `spawn_link(Node, Fun) -> Pid`
  [page 224] Spawn a new process and set a link.
- `spawn_link(Module, Func, Args) -> Pid`
  [page 224] Spawn a new process and set a link.
- `spawn_link(Node, Module, Func, Args) -> Pid`
  [page 224] Spawn a new process and set a link.
- `spawn_opt(Fun, Opt) -> Pid`
  [page 225] Spawn a new process with given options.
- `spawn_opt(Node, Fun, Opt) -> Pid`
  [page 225] Spawn a new process with given options.
- spawn_opt(Module, Func, Args, Opts) -> Pid
  [page 225] Spawn a new process with given options.
- spawn_opt(Node, Module, Func, Args, Opts) -> Pid
  [page 225] Spawn a new process with given options.
- start(Module, Func, Args) -> Ret
  [page 225] Start a new process synchronously.
- start(Module, Func, Args, Time) -> Ret
  [page 225] Start a new process synchronously.
- start(Module, Func, Args, Time, SpawnOpts) -> Ret
  [page 225] Start a new process synchronously.
- start_link(Module, Func, Args) -> Ret
  [page 225] Start a new process synchronously.
- start_link(Module, Func, Args, Time) -> Ret
  [page 225] Start a new process synchronously.
- start_link(Module, Func, Args, Time, SpawnOpts) -> Ret
  [page 225] Start a new process synchronously.
- init_ack(Parent, Ret) -> void()
  [page 226] Used by a process when it has started.
- init_ack(Ret) -> void()
  [page 226] Used by a process when it has started.
- format(CrashReport) -> string()
- initial_call(PidOrPinfo) -> {Module, Function, Args} | Fun | false
  [page 227] Extract the initial call of a proc_lib spawned process.
- translate_initial_call(PidOrPinfo) -> {Module, Function, Arity} | Fun
  [page 227] Extract and translate the initial call of a proc_lib spawned process.

proplists

The following functions are exported:
- append_values(Key, List) -> List
  [page 228]
- compact(List) -> List
  [page 228]
- delete(Key, List) -> List
  [page 228]
- expand(Expansions, List) -> List
  [page 228]
- get_all_values(Key, List) -> [term()]
  [page 229]
- get_bool(Key, List) -> bool()
  [page 229]
- get_keys(List) -> [term()]
  [page 229]
- get_value(Key, List) -> term()
  [page 230]

STDLIB Reference Manual

33
The following functions are exported:

- `cons(Item, Q1) -> Q2` [page 233] Insert an item at the head of a queue
- `daeh(Q) -> Item` [page 233] Return the last item of a queue
- `from_list(L) -> queue()` [page 233] Convert a list to a queue
- `head(Q) -> Item` [page 233] Return the item at the head of a queue
- `in(Item, Q1) -> Q2` [page 233] Insert an item at the tail of a queue
- `in_r(Item, Q1) -> Q2` [page 234] Insert an item at the head of a queue
- `init(Q1) -> Q2` [page 234] Remove the last item from a queue
- `is_empty(Q) -> true | false` [page 234] Test if a queue is empty
- `join(Q1, Q2) -> Q3` [page 234] Join two queues
- `lait(Q1) -> Q2` [page 234] Remove the last item from a queue
The following functions are exported:

- `last(Q) -> Item`
  [page 234] Return the last item of a queue
- `len(Q) -> N`
  [page 234] Get the length of a queue
- `new() -> Q`
  [page 235] Create a new empty FIFO queue
- `out(Q1) -> Result`
  [page 235] Remove the head item from a queue
- `out_r(Q1) -> Result`
  [page 235] Remove the last item from a queue
- `reverse(Q1) -> Q2`
  [page 235] Reverse a queue
- `snoc(Q1, Item) -> Q2`
  [page 235] Insert an item at the end of a queue
- `split(N, Q1) -> {Q2, Q3}`
  [page 235] Split a queue in two
- `tail(Q1) -> Q2`
  [page 235] Remove the head item from a queue
- `to_list(Q) -> list()`
  [page 236] Convert a queue to a list

**random**

The following functions are exported:

- `seed() -> ran()`
  [page 237] Seeds random number generation with default values
- `seed(A1, A2, A3) -> ran()`
  [page 237] Seeds random number generator
- `seed0() -> ran()`
  [page 237] Return default state for random number generation
- `uniform() -> float()`
  [page 237] Return a random float
- `uniform(N) -> int()`
  [page 237] Return a random integer
- `uniform_s(State0) -> {float(), State1}`
  [page 238] Return a random float
- `uniform_s(N, State0) -> {int(), State1}`
  [page 238] Return a random integer

**regexp**

The following functions are exported:

- `match(String, RegExp) -> MatchRes`
  [page 239] Match a regular expression
- `first_match(String, RegExp) -> MatchRes`
  [page 239] Match a regular expression
matches(String, RegExp) -> MatchRes
[page 239] Match a regular expression
sub(String, RegExp, New) -> SubRes
[page 240] Substitute the first occurrence of a regular expression
gsub(String, RegExp, New) -> SubRes
[page 240] Substitute all occurrences of a regular expression
split(String, RegExp) -> SplitRes
[page 240] Split a string into fields
sh_to_awk(ShRegExp) -> AwkRegExp
[page 241] Convert an sh regular expression into an AWK one
parse(RegExp) -> ParseRes
[page 241] Parse a regular expression
format_error(ErrorDescriptor) -> string()
[page 241] Format an error descriptor

sets
The following functions are exported:

new() -> Set
[page 244] Return an empty set
is_set(Set) -> bool()
[page 244] Test for an Set
size(Set) -> int()
[page 244] Return the number of elements in a set
to_list(Set) -> List
[page 244] Convert an Set into a list
from_list(List) -> Set
[page 244] Convert a list into an Set
is_element(Element, Set) -> bool()
[page 244] Test for membership of an Set
add_element(Element, Set1) -> Set2
[page 245] Add an element to an Set
del_element(Element, Set1) -> Set2
[page 245] Remove an element from an Set
union(Set1, Set2) -> Set3
[page 245] Return the union of two Sets
union(SetList) -> Set
[page 245] Return the union of a list of Sets
intersection(Set1, Set2) -> Set3
[page 245] Return the intersection of two Sets
intersection(SetList) -> Set
[page 245] Return the intersection of a list of Sets
subtract(Set1, Set2) -> Set3
[page 245] Return the difference of two Sets
is_subset(Set1, Set2) -> bool()
[page 246] Test for subset
- fold(Function, Acc0, Set) -> Acc1  
  [page 246] Fold over set elements
- filter(Pred, Set) -> Set2  
  [page 246] Filter set elements

shell

The following functions are exported:

- history(N) -> integer()  
  [page 254] Sets the number of previous commands to keep
- results(N) -> integer()  
  [page 254] Sets the number of previous commands to keep
- start_restricted(Module) -> ok  
  [page 254] Exits a normal shell and starts a restricted shell.
- stop_restricted(Module) -> ok  
  [page 254] Exits a restricted shell and starts a normal shell.

shell_default

No functions are exported.

slave

The following functions are exported:

- start(Host)  
  [page 256] Start a slave node at Host
- start_link(Host)  
  [page 256] Start a slave node at Host
- start(Host, Name)  
  [page 256] Start a slave node at Host called Name@Host
- start_link(Host, Name)  
  [page 257] Start a slave node at Host called Name@Host
- start(Host, Name, Args) -> {ok, Node} | {error, ErrorInfo}  
  [page 257] Start a slave node at Host called Name@Host and passes Args to new node
- start_link(Host, Name, Args)  
  [page 257] Start a slave node at Host called Name@Host
- stop(Node)  
  [page 258] Stop (kill) a node
- pseudo([Master | ServerList])  
  [page 258] Start a number of pseudo servers
- pseudo(Master, ServerList)  
  [page 258] Start a number of pseudo servers
- relay(Pid)  
  [page 258] Run a pseudo server
sofs

The following functions are exported:

- `a_function(Tuples [, Type]) -> Function` [page 263] Create a function.
- `canonical_relation(SetOfSets) -> BinRel` [page 263] Return the canonical map.
- `composite(Function1, Function2) -> Function3` [page 263] Return the composite of two functions.
- `constant_function(Set, AnySet) -> Function` [page 263] Create the function that maps each element of a set onto another set.
- `converse(BinRel1) -> BinRel2` [page 264] Return the converse of a binary relation.
- `difference(Set1, Set2) -> Set3` [page 264] Return the difference of two sets.
- `digraph_to_family(Graph [, Type]) -> Family` [page 264] Create a family from a directed graph.
- `domain(BinRel) -> Set` [page 264] Return the domain of a binary relation.
- `drestriction(BinRel1, Set) -> BinRel2` [page 264] Return a restriction of a binary relation.
- `drestriction(SetFun, Set1, Set2) -> Set3` [page 265] Return a restriction of a relation.
- `empty_set() -> Set` [page 265] Return the untyped empty set.
- `extension(BinRel1, Set, AnySet) -> BinRel2` [page 265] Extend the domain of a binary relation.
- `family(Tuples [, Type]) -> Family` [page 266] Create a family of subsets.
- `family_difference(Family1, Family2) -> Family3` [page 266] Return the difference of two families.
- `family_domain(Family1) -> Family2` [page 266] Return a family of domains.
- `family_field(Family1) -> Family2` [page 266] Return a family of fields.
- `family_intersection(Family1) -> Family2` [page 267] Return the intersection of a family of sets of sets.
- `family_intersection(Family1, Family2) -> Family3` [page 267] Return the intersection of two families.
- `family_projection(SetFun, Family1) -> Family2` [page 267] Return a family of modified subsets.
- `family_range(Family1) -> Family2` [page 267] Return a family of ranges.
- `family_specification(Fun, Family1) -> Family2` [page 268] Select a subset of a family using a predicate.
- `family_to_digraph(Family [, GraphType])` -> `Graph` [page 268] Create a directed graph from a family.
- `family_to_relation(Family)` -> `BinRel` [page 268] Create a binary relation from a family.
- `family_union(Family1)` -> `Family2` [page 269] Return the union of a family of sets of sets.
- `family_union(Family1, Family2)` -> `Family3` [page 269] Return the union of two families.
- `field(BinRel)` -> `Set` [page 269] Return the field of a binary relation.
- `from_external(ExternalSet, Type)` -> `AnySet` [page 269] Create a set.
- `from_sets(ListOfSets)` -> `Set` [page 269] Create a set out of a list of sets.
- `from_sets(TupleOfSets)` -> `Ordset` [page 269] Create an ordered set out of a tuple of sets.
- `from_term(Term [, Type])` -> `AnySet` [page 270] Create a set.
- `image(BinRel, Set1)` -> `Set2` [page 271] Return the image of a set under a binary relation.
- `intersection(SetOfSets)` -> `Set` [page 271] Return the intersection of a set of sets.
- `intersection(Set1, Set2)` -> `Set3` [page 271] Return the intersection of two sets.
- `intersection_of_family(Family)` -> `Set` [page 271] Return the intersection of a family.
- `inverse(Function1)` -> `Function2` [page 271] Return the inverse of a function.
- `inverse_image(BinRel, Set1)` -> `Set2` [page 272] Return the inverse image of a set under a binary relation.
- `is_a_function(BinRel)` -> `Bool` [page 272] Test for a function.
- `is_disjoint(Set1, Set2)` -> `Bool` [page 272] Test for disjoint sets.
- `is_empty_set(AnySet)` -> `Bool` [page 272] Test for an empty set.
- `is_equal(AnySet1, AnySet2)` -> `Bool` [page 272] Test two sets for equality.
- `is_set(AnySet)` -> `Bool` [page 272] Test for an unordered set.
- `is_sofs_set(Term)` -> `Bool` [page 273] Test for a type.
join(Relation1, I, Relation2, J) -> Relation3
[page 273] Return the join of two relations.

multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2
[page 273] Return the multiple relative product of a tuple of binary relations and a relation.

no_elements(ASet) -> NoElements
[page 274] Return the number of elements of a set.

partition(SetOfSets) -> Partition
[page 274] Return the coarsest partition given a set of sets.

partition(SetFun, Set) -> Partition
[page 274] Return a partition of a set.

partition(SetFun, Set1, Set2) -> {Set3, Set4}
[page 274] Return a partition of a set.

partition_family(SetFun, Set) -> Family
[page 275] Return a family indexing a partition.

product(TupleOfSets) -> Relation
[page 275] Return the Cartesian product of a tuple of sets.

product(Set1, Set2) -> BinRel
[page 275] Return the Cartesian product of two sets.

projection(SetFun, Set1) -> Set2
[page 276] Return a set of substituted elements.

range(BinRel) -> Set
[page 276] Return the range of a binary relation.

relation(Tuples [, Type]) -> Relation
[page 276] Create a relation.

relation_to_family(BinRel) -> Family
[page 276] Create a family from a binary relation.

relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2
[page 277] Return the relative product of a tuple of binary relations and a binary relation.

relative_product(BinRel1, BinRel2) -> BinRel3
[page 277] Return the relative product of two binary relations.

relative_product1(BinRel1, BinRel2) -> BinRel3
[page 277] Return the relative product of two binary relations.

restriction(BinRel1, Set) -> BinRel2
[page 277] Return a restriction of a binary relation.

restriction(SetFun, Set1, Set2) -> Set3
[page 278] Return a restriction of a set.

set(Terms [, Type]) -> Set
[page 278] Create a set of atoms or any type of sets.

specification(Fun, Set1) -> Set2
[page 278] Select a subset using a predicate.

strict_relation(BinRel1) -> BinRel2
[page 278] Return the strict relation corresponding to a given relation.

substitution(SetFun, Set1) -> Set2
[page 279] Return a function with a given set as domain.
The following functions are exported:

- `symdiff(Set1, Set2) -> Set3`  
  [page 279] Return the symmetric difference of two sets.
- `symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`  
  [page 280] Return a partition of two sets.
- `to_external(AnySet) -> ExternalSet`  
  [page 280] Return the elements of a set.
- `to_sets(ASet) -> Sets`  
  [page 280] Return a list or a tuple of the elements of set.
- `type(AnySet) -> Type`  
  [page 280] Return the type of a set.
- `union(SetOfSets) -> Set`  
  [page 280] Return the union of a set of sets.
- `union(Set1, Set2) -> Set3`  
  [page 280] Return the union of two sets.
- `union_of_family(Family) -> Set`  
  [page 281] Return the union of a family.
- `weak_relation(BinRel1) -> BinRel2`  
  [page 281] Return the weak relation corresponding to a given relation.

**string**

The following functions are exported:

- `len(String) -> Length`  
  [page 282] Return the length of a string
- `equal(String1, String2) -> bool()`  
  [page 282] Test string equality
- `concat(String1, String2) -> String3`  
  [page 282] Concatenate two strings
- `chr(String, Character) -> Index`  
  [page 282] Return the index of the first/last occurrence of Character in String
- `rchr(String, Character) -> Index`  
  [page 282] Return the index of the first/last occurrence of Character in String
- `str(String, SubString) -> Index`  
  [page 282] Find the index of a substring
- `rstr(String, SubString) -> Index`  
  [page 282] Find the index of a substring
- `span(String, Chars) -> Length`  
  [page 283] Span characters at start of string
- `cspan(String, Chars) -> Length`  
  [page 283] Span characters at start of string
- `substr(String, Start) -> SubString`  
  [page 283] Return a substring of String
- `substr(String, Start, Length) -> Substring`  
  [page 283] Return a substring of String
- `tokens(String, SeparatorList) -> Tokens`  
  [page 283] Split string into tokens
- `chars(Character, Number) -> String`  
  [page 283] Returns a string consisting of numbers of characters
- `chars(Character, Number, Tail) -> String`  
  [page 283] Returns a string consisting of numbers of characters
- `copies(String, Number) -> Copies`  
  [page 284] Copy a string
- `words(String) -> Count`  
  [page 284] Count blank separated words
- `words(String, Character) -> Count`  
  [page 284] Count blank separated words
- `sub_word(String, Number) -> Word`  
  [page 284] Extract subword
- `sub_word(String, Number, Character) -> Word`  
  [page 284] Extract subword
- `strip(String) -> Stripped`  
  [page 284] Strip leading or trailing characters
- `strip(String, Direction) -> Stripped`  
  [page 284] Strip leading or trailing characters
- `strip(String, Direction, Character) -> Stripped`  
  [page 284] Strip leading or trailing characters
- `left(String, Number) -> Left`  
  [page 285] Adjust left end of string
- `left(String, Number, Character) -> Left`  
  [page 285] Adjust left end of string
- `right(String, Number) -> Right`  
  [page 285] Adjust right end of string
- `right(String, Number, Character) -> Right`  
  [page 285] Adjust right end of string
- `centre(String, Number) -> Centered`  
  [page 285] Center a string
- `centre(String, Number, Character) -> Centered`  
  [page 285] Center a string
- `sub_string(String, Start) -> SubString`  
  [page 285] Extract a substring
- `sub_string(String, Start, Stop) -> SubString`  
  [page 286] Extract a substring

**supervisor**

The following functions are exported:

- `start_link(Module, Args) -> Result`  
  [page 289] Create a supervisor process.
- `start_link(SupName, Module, Args) -> Result`  
  [page 289] Create a supervisor process.
- `start_child(SupRef, ChildSpec) -> Result`  
  [page 289] Dynamically add a child process to a supervisor.
- `terminate_child(SupRef, Id) -> Result`
  [page 290] Terminate a child process belonging to a supervisor.
- `delete_child(SupRef, Id) -> Result`
  [page 291] Delete a child specification from a supervisor.
- `restart_child(SupRef, Id) -> Result`
  [page 291] Restart a terminated child process belonging to a supervisor.
- `which_children(SupRef) -> [Id, Child, Type, Modules]`
  [page 292] Return information about all children specifications and child processes belonging to a supervisor.
- `check_childspecs([ChildSpec]) -> Result`
  [page 292] Check if child specifications are syntactically correct.
- `Module:init(Args) -> Result`
  [page 293] Return a supervisor specification.

**supervisor_bridge**

The following functions are exported:

- `start_link(Module, Args) -> Result`
  [page 294] Create a supervisor bridge process.
- `start_link(SupBridgeName, Module, Args) -> Result`
  [page 294] Create a supervisor bridge process.
- `Module:init(Args) -> Result`
  [page 295] Initialize process and start subsystem.
- `Module:terminate(Reason, State)`
  [page 295] Clean up and stop subsystem.

**SYS**

The following functions are exported:

- `log(Name,Flag)`
  [page 298] Log system events in memory
- `log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}`
  [page 298] Log system events in memory
- `log_to_file(Name,Flag)`
  [page 298] Log system events to the specified file
- `log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`
  [page 298] Log system events to the specified file
- `statistics(Name,Flag)`
  [page 298] Enable or disable the collections of statistics
- `statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`
  [page 298] Enable or disable the collections of statistics
- `trace(Name,Flag)`
  [page 299] Print all system events on standard_io
- `trace(Name,Flag,Timeout) -> void()`
  [page 299] Print all system events on standard_io
- `no_debug(Name)`
  [page 299] Turn off debugging

- `no_debug(Name,Timeout) -> void()`
  [page 299] Turn off debugging

- `suspend(Name)`
  [page 299] Suspend the process

- `suspend(Name,Timeout) -> void()`
  [page 299] Suspend the process

- `resume(Name)`
  [page 299] Resume a suspended process

- `resume(Name,Timeout) -> void()`
  [page 299] Resume a suspended process

- `change_code(Name, Module, OldVsn, Extra)`
  [page 299] Send the code change system message to the process

- `change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}`
  [page 299] Send the code change system message to the process

- `get_status(Name)`
  [page 299] Get the status of the process

- `get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`
  [page 299] Get the status of the process

- `install(Name,Func,FuncState)`
  [page 300] Install a debug function in the process

- `install(Name,Func,FuncState,Timeout)`
  [page 300] Install a debug function in the process

- `remove(Name,Func)`
  [page 300] Remove a debug function from the process

- `remove(Name,Func,Timeout) -> void()`
  [page 300] Remove a debug function from the process

- `debug_options(Options) -> [dbg_opt()]`
  [page 301] Convert a list of options to a debug structure

- `get_debug(Item,Debug,Default) -> term()`
  [page 301] Get the data associated with a debug option

- `handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]`
  [page 301] Generate a system event

- `handle_system_msg(Msg,From,Parent,Module,Debug,Misc)`
  [page 301] Take care of system messages

- `print_log(Debug) -> void()`
  [page 302] Print the logged events in the debug structure

- `Mod:system_continue(Parent, Debug, Misc)`
  [page 302] Called when the process should continue its execution

- `Mod:system_terminate(Reason, Parent, Debug, Misc)`
  [page 302] Called when the process should terminate

- `Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}`
  [page 302] Called when the process should perform a code change
timer

The following functions are exported:

- `start() -> ok`
  [page 304] Start a global timer server (named timer_server).
- `apply_after(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`
- `send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 304] Send Message to Pid after a specified Time.
- `send_after(Time, Message) -> {ok, TRef} | {error, Reason}`
  [page 304] Send Message to Pid after a specified Time.
- `exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error, Reason2}`
  [page 305] Send an exit signal with Reason after a specified Time.
- `exit_after(Time, Reason1) -> {ok, TRef} | {error, Reason2}`
  [page 305] Send an exit signal with Reason after a specified Time.
- `kill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}`
  [page 305] Send an exit signal with Reason after a specified Time.
- `kill_after(Time) -> {ok, TRef} | {error, Reason2}`
  [page 305] Send an exit signal with Reason after a specified Time.
- `apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`
  [page 305] Evaluate Module:Function(Arguments) repeatedly at intervals of Time.
- `send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 305] Send Message repeatedly at intervals of Time.
- `send_interval(Time, Message) -> {ok, TRef} | {error, Reason}`
  [page 305] Send Message repeatedly at intervals of Time.
- `cancel(TRef) -> {ok, cancel} | {error, Reason}`
  [page 305] Cancel a previously requested timeout identified by TRef.
- `sleep(Time) -> ok`
  [page 305] Suspend the calling process for Time amount of milliseconds.
- `tc(Module, Function, Arguments) -> {Time, Value}`
  [page 306] Measure the real time it takes to evaluate apply(Module, Function, Arguments)
- `now_diff(T2, T1) -> {Time, Value}`
  [page 306] Calculate time difference between now/0 timestamps
- `seconds(Seconds) -> Milliseconds`
- `minutes(Minutes) -> Milliseconds`
  [page 306] Converts Minutes to Milliseconds.
- `hours(Hours) -> Milliseconds`
  [page 306] Convert Hours to Milliseconds.
- `hms(Hours, Minutes, Seconds) -> Milliseconds`
  [page 306] Convert Hours+Minutes+Seconds to Milliseconds.
The following functions are exported:

- `change_key(RegHandle, Key) -> ReturnValue`
  
  Move to a key in the registry

- `change_key_create(RegHandle, Key) -> ReturnValue`
  
  Move to a key, create it if it is not there

- `close(RegHandle) -> ReturnValue`
  
  Close the registry.

- `current_key(RegHandle) -> ReturnValue`
  
  Return the path to the current key.

- `delete_key(RegHandle) -> ReturnValue`
  
  Delete the current key

- `delete_value(RegHandle, Name) -> ReturnValue`
  
  Delete the named value on the current key.

- `expand(String) -> ExpandedString`
  
  Expand a string with environment variables

- `format_error(ErrorId) -> ErrorString`
  
  Convert a POSIX error code to a string

- `open(OpenModeList) -> ReturnValue`
  
  Open the registry for reading or writing

- `set_value(RegHandle, Name, Value) -> ReturnValue`
  
  Set value at the current registry key with specified name.

- `sub_keys(RegHandle) -> ReturnValue`
  
  Get subkeys to the current key.

- `value(RegHandle, Name) -> ReturnValue`
  
  Get the named value on the current key.

- `values(RegHandle) -> ReturnValue`
  
  Get all values on the current key.
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 \texttt{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 \texttt{chunks/2} -

The \texttt{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 \texttt{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.

```erlang
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`).

```erlang
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.

```erlang
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.

```erlang
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.

\[
\text{strip_release}(	ext{Directory}) \rightarrow \{ \text{ok, \{Module, FileName\}} \} | \{ \text{error, Module, Reason} \}
\]

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.

\[
\text{format_error}(\text{Error}) \rightarrow \text{character_list()}
\]

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 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.

\begin{verbatim}
\textbf{m(Module)} \rightarrow \textbf{void()}
Types:
- Module = \textbf{atom()}
This function lists information about Module.
\end{verbatim}

\begin{verbatim}
\textbf{memory()} \rightarrow \textbf{MemoryInformation}
The same as \texttt{erlang:memory/0}, see the \texttt{erlang(3)} man page.
\end{verbatim}

\begin{verbatim}
\textbf{memory(MemoryTypeSpecification)} \rightarrow \textbf{MemoryInformation}
The same as \texttt{erlang:memory/1}, see the \texttt{erlang(3)} man page.
\end{verbatim}

\begin{verbatim}
\textbf{nc(File)} \rightarrow \textbf{void()}
Types:
- File = \textbf{atom()} \mid \textbf{string()}
This function compiles File and loads it on all nodes in an Erlang nodes network.
\end{verbatim}

\begin{verbatim}
\textbf{nc(File, Flags)} \rightarrow \textbf{void()}
Types:
- File = \textbf{atom()} \mid \textbf{string()}
- Flags = [\textbf{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.
\end{verbatim}

\begin{verbatim}
\textbf{ni()} \rightarrow \textbf{void()}
This function does the same as \texttt{i()}, but for all nodes in the network.
\end{verbatim}

\begin{verbatim}
\textbf{nl(Module)} \rightarrow \textbf{void()}
Types:
- Module = \textbf{atom()}
This function loads Module on all nodes in an Erlang nodes network.
\end{verbatim}

\begin{verbatim}
\textbf{nregs()} \rightarrow \textbf{void()}
This function is the same as \texttt{regs()}, but on all nodes in the system.
\end{verbatim}

\begin{verbatim}
\textbf{pid(X, Y, Z)} \rightarrow \textbf{pid()}
Types:
- X = Y = Z = \textbf{int()}
\end{verbatim}
This function converts the integers \( X, Y, \) and \( Z \) to the Pid \( \langle X \cdot Y \cdot Z \rangle \). It saves typing and the use of `list_to_pid/1`. This function should only be used when debugging.

\[ \text{pwd() \to void()} \]
This function prints the current working directory.

\[ \text{q() \to void()} \]
This function is shorthand for \texttt{init:stop()}, i.e., it causes the node to stop in a controlled fashion.

\[ \text{regs() \to void()} \]
This function displays formatted information about all registered processes in the system.

\[ \text{xm(ModSpec) \to void()} \]
Types:
- \( \text{ModSpec} = \text{Module} | \text{File} \)
- \( \text{Module} = \text{atom}() \)
- \( \text{File} = \text{string}() \)
This function finds undefined functions and unused functions in a module by calling \texttt{xref:m/1}.

\[ \text{zi() \to void()} \]
This function works like \texttt{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)
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.

\[ \text{local} \_\text{time}() \rightarrow \{\text{Date, Time}\} \]

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.

\[ \text{local} \_\text{time} \_\text{to} \_\text{universal} \_\text{time}(\{\text{Date, Time}\}) \rightarrow \{\text{Date, Time}\} \]

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 \text{local} \_\text{time} \_\text{to} \_\text{universal} \_\text{time} \_\text{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.

\[ \text{local} \_\text{time} \_\text{to} \_\text{universal} \_\text{time} \_\text{dst}(\text{DateTime}) \rightarrow [\text{DateTimeUTC}] \]

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.

\[ \text{now} \_\text{to} \_\text{local} \_\text{time}(\text{Now}) \rightarrow \{\text{Date, Time}\} \]
Types:
- \( \text{Now} = \{ \text{MegaSecs}, \text{Secs}, \text{MicroSecs} \} \)
- \( \text{Date} = \{ \text{Year}, \text{Month}, \text{Day} \} \)
- \( \text{Time} = \{ \text{Hour}, \text{Minute}, \text{Second} \} \)
- \( \text{MegaSecs} = \text{Secs} = \text{Millisecs} = \text{int()} \)
- \( \text{Year} = \text{Month} = \text{Day} = \text{Hour} = \text{Minute} = \text{Second} = \text{int()} \)

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

\[
\text{now} \rightarrow (\text{Date}, \text{Time})
\]

\[
\text{now} \rightarrow (\text{Date}, \text{Time})
\]

Types:
- \( \text{Now} = \{ \text{MegaSecs}, \text{Secs}, \text{MicroSecs} \} \)
- \( \text{Date} = \{ \text{Year}, \text{Month}, \text{Day} \} \)
- \( \text{Time} = \{ \text{Hour}, \text{Minute}, \text{Second} \} \)
- \( \text{MegaSecs} = \text{Secs} = \text{Millisecs} = \text{int()} \)
- \( \text{Year} = \text{Month} = \text{Day} = \text{Hour} = \text{Minute} = \text{Second} = \text{int()} \)

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

\[
\text{seconds} \rightarrow (\text{Days}, \text{Time})
\]

Types:
- \( \text{Time()} = \{ \text{Hour}, \text{Minute}, \text{Second} \} \)
- \( \text{Hour} = \text{Minute} = \text{Second} = \text{Days} = \text{int()} \)

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.

\[
\text{seconds} \rightarrow \text{Time}
\]

Types:
- \( \text{Time()} = \{ \text{Hour}, \text{Minute}, \text{Second} \} \)
- \( \text{Hour} = \text{Minute} = \text{Second} = \text{Secs} = \text{int()} \)

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

\[
\text{time} \rightarrow \text{Tdiff}
\]

Types:
- \( \text{T1} = \text{T2} = \{ \text{Date}, \text{Time} \} \)
- \( \text{Tdiff} = \{ \text{Day}, \{ \text{Hour}, \text{Minute}, \text{Second} \} \} \)
- \( \text{Date} = \{ \text{Year}, \text{Month}, \text{Day} \} \)
- \( \text{Time} = \{ \text{Hour}, \text{Minute}, \text{Second} \} \)
- \( \text{Year} = \text{Month} = \text{Day} = \text{Hour} = \text{Minute} = \text{Second} = \text{int()} \)
This function returns the difference between two \{Date, Time\} structures. \(T_2\) should refer to an epoch later than \(T_1\).

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

time_to_seconds(Time) \to Secs

Types:
- \(\text{Time} = \{\text{Hour}, \text{Minute}, \text{Second}\}\)
- \(\text{Hour} = \text{Minute} = \text{Second} = \text{Secs} = \text{int}()\)

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

universal_time() \to \{Date, Time\}

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

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]) \to \{Date, Time\}

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

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

valid_date(Date) \to \text{bool}()
valid_date(Year, Month, Day) \to \text{bool}()

Types:
- \(\text{Date} = \{\text{Year}, \text{Month}, \text{Day}\}\)
- \(\text{Year} = \text{Month} = \text{Day} = \text{int}()\)

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.
The module \texttt{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.

\textbf{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 \texttt{first} and \texttt{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()
noslots() = 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_fixmap/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 \((\text{Continuation2}, \text{Data})\), 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 \((\text{error}, \text{Reason})\) if an error occurs.

\[
\text{close}(\text{Name}) \rightarrow \text{ok} | \{\text{error}, \text{Reason}\}
\]

Types:

- \(\text{Name} = \text{name()}\)

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.

\[
\text{delete}(\text{Name}, \text{Key}) \rightarrow \text{ok} | \{\text{error}, \text{Reason}\}
\]

Types:

- \(\text{Name} = \text{name()}\)

Deletes all objects with the key \(\text{Key}\) from the table \(\text{Name}\).

\[
\text{delete_all_objects}(\text{Name}) \rightarrow \text{ok} | \{\text{error}, \text{Reason}\}
\]

Types:

- \(\text{Name} = \text{name()}\)

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

\[
\text{delete_object}(\text{Name}, \text{Object}) \rightarrow \text{ok} | \{\text{error}, \text{Reason}\}
\]

Types:

- \(\text{Name} = \text{name()}\)
- \(\text{Object} = \text{object()}\)

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.

\[
\text{first}(\text{Name}) \rightarrow \text{Key} | \text{'$end_of_table'}
\]

Types:
Returns the first key stored in the table \texttt{Name} according to the table's internal order, or \texttt{'$end\_of\_table'} if the table is empty.

Unless the table is protected using \texttt{safe\_fixtable/2}, subsequent calls to \texttt{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 \{\texttt{error, Reason}\}. The reason for not returning the error tuple is that it cannot be distinguished from a key.

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

\begin{verbatim}
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 \texttt{Name} together with an extra argument \texttt{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. \texttt{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 \texttt{Name} together with an extra argument \texttt{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. \texttt{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 \texttt{Name} with the objects of the Ets table \texttt{EtsTab}. The order in which the objects are inserted is not specified. Since \texttt{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(Args) -> Res
- **Args** = 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**.

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.

```prolog
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.

```prolog
is_compatible_bchunk_format(Name, BchunkFormat) -> Bool
```

**Types:**
is_dets_file(FileName) -> Bool | {error, Reason}

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

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

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}

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:
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 116] 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 116] 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 of 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 116] 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 116] 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.

  \[
  \text{next(Name, Key1)} \rightarrow \text{Key2} \mid \text{"$end_of_table"}
  \]

  **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.

  \[
  \text{open_file(Filename)} \rightarrow \{\text{ok, Reference}\} \mid \{\text{error, Reason}\}
  \]

  **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.

  \[
  \text{open_file(Name, Args)} \rightarrow \{\text{ok, Name}\} \mid \{\text{error, Reason}\}
  \]

  **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.

  \[
  \text{Dets STDLIB Reference Manual}
  \]
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.

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.

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.

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.

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.

The type of the table. The default value is set.

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 \texttt{Fix} is \texttt{true}, the table \texttt{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 \texttt{first/1}, \texttt{next/2}, \texttt{select} and \texttt{match} functions work as expected even if the table has been fixed; the limited support for concurrency implemented in Ets has not yet been implemented 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.

\begin{verbatim}
select(Continuation) -> \{Selection, Continuation2\} \{error, Reason\}
Types:
  • Continuation = Continuation2 = select_cont()
  • Selection = [\text{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 \texttt{select/1} or \texttt{select/3}.

When all objects of the table have been matched, \'\$end_of_table' \ is returned.
\end{verbatim}

\begin{verbatim}
select(Name, MatchSpec) -> Selection \{error, Reason\}
Types:
  • Name = \text{name()}
  • MatchSpec = \text{match_spec()}
  • Selection = [\text{term()}]
Returns the results of applying the match specification \texttt{MatchSpec} to all or some objects stored in the table \texttt{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 \texttt{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 \texttt{select} functions for traversing all objects of a table is more efficient than calling \texttt{first/1} and \texttt{next/2} of \texttt{slot/2}.
\end{verbatim}

\begin{verbatim}
select(Name, MatchSpec, N) -> \{Selection, Continuation\} \{error, Reason\}
Types:
  • Name = \text{name()}
  • MatchSpec = \text{match_spec()}
  • N = default \ | \text{int()}
  • Selection = [\text{term()}]
  • Continuation = select_cont()
\end{verbatim}
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()`

Delete 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}
```

---

STDLIB Reference Manual

73
traverse(Name, Fun) -> Result | {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 function 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 116], 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()

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(Func, D1, D2) ->
    fold(fun (K, V1, D) ->
        update(K, fun (V2) -> Func(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:

```erlang
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:

```erlang
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:

```erlang
> 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 149], orddict(3) [page 219]
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 \(V \times V\) (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 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 \(1 \leq i < k\). 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:
- \(G = \text{digraph}()\)
- \(E = \text{edge}()\)
- \(V1 = V2 = \text{vertex}()\)
- \(\text{Label} = \text{label}()\)
- \(\text{Reason} = \{\text{bad_edge}, \text{Path}\} | \{\text{bad_vertex}, V\}\)
- \(\text{Path} = [\text{vertex}()]\)
add_edge/5 creates (or modifies) the edge \( E \) of the digraph \( G \), using Label as the (new) label [page 80] of the edge. The edge is emanating [page 80] from \( V_1 \) and incident [page 80] on \( V_2 \). Returns \( E \).

\( \text{add_edge}(G, V_1, V_2, \text{Label}) \) is equivalent to \( \text{add_edge}(G, E, V_1, V_2, \text{Label}) \), where \( E \) is a created edge. Tuples on the form \(['e' | N]\), where \( N \) is an integer \( \geq 1 \), are used for representing the created edges.

\( \text{add_edge}(G, V_1, V_2) \) is equivalent to \( \text{add_edge}(G, V_1, V_2, []) \).

If the edge would create a cycle in an acyclic digraph [page 80], then \{error, \{bad_edge, Path\}\} is returned. If either of \( V_1 \) or \( V_2 \) is not a vertex of the digraph \( G \), then \{error, \{bad_vertex, V\}\} is returned, \( V = V_1 \) or \( V = V_2 \).

\( \text{add_vertex}(G, V, \text{Label}) \rightarrow \text{vertex}() \)

\( \text{add_vertex}(G, V) \rightarrow \text{vertex}() \)

\( \text{add_vertex}(G) \rightarrow \text{vertex}() \)

Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)
- \( \text{Label} = \text{label}() \)

\( \text{add_vertex}/3 \) creates (or modifies) the vertex \( V \) of the digraph \( G \), using Label as the (new) label [page 80] of the vertex. Returns \( V \).

\( \text{add_vertex}(G, V) \) is equivalent to \( \text{add_vertex}(G, V, []) \).

\( \text{add_vertex}/1 \) 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 \( \geq 1 \), are used for representing the created vertices.

\( \text{del_edge}(G, E) \rightarrow \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( E = \text{edge}() \)

Deletes the edge \( E \) from the digraph \( G \).

\( \text{del_edges}(G, \text{Edges}) \rightarrow \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( \text{Edges} = \text{[edge()]} \)

Deletes the edges in the list \( \text{Edges} \) from the digraph \( G \).

\( \text{del_path}(G, V_1, V_2) \rightarrow \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( V_1 = V_2 = \text{vertex}() \)
Deletes edges from the digraph \( G \) until there are no paths [page 80] from the vertex \( V_1 \) to the vertex \( V_2 \).

A sketch of the procedure employed: Find an arbitrary simple path [page 80] \( v[1], v[2], ..., v[k] \) from \( V_1 \) to \( V_2 \) in \( G \). Remove all edges of \( G \) emanating [page 80] from \( v[i] \) and incident [page 80] to \( v[i+1] \) for \( 1 \leq i < k \) (including multiple edges). Repeat until there is no path between \( V_1 \) and \( V_2 \).

\[ \text{del_vertex}(G, V) \rightarrow \text{true} \]

Types:
- \( G = \text{digraph()} \)
- \( V = \text{vertex()} \)

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating [page 80] from \( V \) or incident [page 80] on \( V \) are also deleted.

\[ \text{del_vertices}(G, \text{Vertices}) \rightarrow \text{true} \]

Types:
- \( G = \text{digraph()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Deletes the vertices in the list \( \text{Vertices} \) from the digraph \( G \).

\[ \text{delete}(G) \rightarrow \text{true} \]

Types:
- \( G = \text{digraph()} \)

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

\[ \text{edge}(G, E) \rightarrow \{E, V_1, V_2, \text{Label}\} | \text{false} \]

Types:
- \( G = \text{digraph()} \)
- \( E = \text{edge()} \)
- \( V_1 = V_2 = \text{vertex()} \)
- \( \text{Label} = \text{label()} \)

Returns \( \{E, V_1, V_2, \text{Label}\} \) where \( \text{Label} \) is the label [page 80] of the edge \( E \) emanating [page 80] from \( V_1 \) and incident [page 80] on \( V_2 \) of the digraph \( G \). If there is no edge \( E \) of the digraph \( G \), then false is returned.

\[ \text{edges}(G) \rightarrow \text{Edges} \]

Types:
- \( G = \text{digraph()} \)
- \( \text{Edges} = [\text{edge()}] \)

Returns a list of all edges of the digraph \( G \), in some unspecified order.

\[ \text{edges}(G, V) \rightarrow \text{Edges} \]
Types:
- $G = \text{digraph}()$
- $V = \text{vertex}()$
- $\text{Edges} = [\text{edge}()]$

Returns a list of all edges emanating [page 80] from or incident [page 80] on $V$ of the digraph $G$, in some unspecified order.

$\text{get\_cycle}(G, V) \rightarrow \text{Vertices} | \text{false}$

Types:
- $G = \text{digraph}()$
- $V1 = V2 = \text{vertex}()$
- $\text{Vertices} = [\text{vertex}()]$

If there is a simple cycle [page 80] of length two or more through the vertex $V$, then the cycle is returned as a list $[V, \ldots, V]$ of vertices, otherwise if there is a loop [page 80] through $V$, then the loop is returned as a list $[V]$. If there are no cycles through $V$, then false is returned.

$\text{get\_path}/3$ is used for finding a simple cycle through $V$.

$\text{get\_path}(G, V1, V2) \rightarrow \text{Vertices} | \text{false}$

Types:
- $G = \text{digraph}()$
- $V1 = V2 = \text{vertex}()$
- $\text{Vertices} = [\text{vertex}()]$

Tries to find a simple path [page 80] from the vertex $V1$ to the vertex $V2$ of the digraph $G$. Returns the path as a list $[V1, \ldots, V2]$ 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.

$\text{get\_short\_cycle}(G, V) \rightarrow \text{Vertices} | \text{false}$

Types:
- $G = \text{digraph}()$
- $V1 = V2 = \text{vertex}()$
- $\text{Vertices} = [\text{vertex}()]$

Tries to find an as short as possible simple cycle [page 80] through the vertex $V$ of the digraph $G$. Returns the cycle as a list $[V, \ldots, V]$ of vertices, or false if no simple cycle through $V$ exists. Note that a loop [page 80] through $V$ is returned as the list $[V, V]$.

$\text{get\_short\_path}/3$ is used for finding a simple cycle through $V$.

$\text{get\_short\_path}(G, V1, V2) \rightarrow \text{Vertices} | \text{false}$

Types:
- $G = \text{digraph}()$
- $V1 = V2 = \text{vertex}()$
- $\text{Vertices} = [\text{vertex}()]$
Tries to find an as short as possible simple path [page 80] from the vertex $V_1$ to the vertex $V_2$ of the digraph $G$. Returns the path as a list $[V_1, \ldots, V_2]$ of vertices, or false if no simple path from $V_1$ to $V_2$ of length one or more exists.

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

\[
in\text{\_degree}(G, V) \rightarrow \text{integer()}
\]

Types:
- $G = \text{digraph()}$
- $V = \text{vertex()}$

Returns the in-degree [page 80] of the vertex $V$ of the digraph $G$.

\[
in\text{\_edges}(G, V) \rightarrow \text{Edges}
\]

Types:
- $G = \text{digraph()}$
- $V = \text{vertex()}$
- $\text{Edges} = [\text{edge()}]$

Returns a list of all edges incident [page 80] on $V$ of the digraph $G$, in some unspecified order.

\[
in\text{\_neighbours}(G, V) \rightarrow \text{Vertices}
\]

Types:
- $G = \text{digraph()}$
- $V = \text{vertex()}$
- $\text{Vertices} = [\text{vertex()}]$

Returns a list of all in-neighbours [page 80] of $V$ of the digraph $G$, in some unspecified order.

\[
\text{info}(G) \rightarrow \text{InfoList}
\]

Types:
- $G = \text{digraph()}$
- $\text{InfoList} = [\text{cyclicity, Cyclicity}, \text{memory, NoWords}, \text{protection, Protection}]$
- $\text{Cyclicity} = \text{cyclic} \mid \text{acyclic}$
- $\text{Protection} = \text{public} \mid \text{protected} \mid \text{private}$
- $\text{NoWords} = \text{integer()} \geq 0$

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

- $\{\text{cyclicity, Cyclicity}\}$, where Cyclicity is cyclic or acyclic, according to the options given to new.
- $\{\text{memory, NoWords}\}$, where NoWords is the number of words allocated to the sets tables.
- $\{\text{protection, Protection}\}$, 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 80] with properties according to the options in Type:
   cyclic  Allow cycles [page 80] in the digraph (default).
   acyclic  The digraph is to be kept acyclic [page 80].
   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 80] 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 80] from V of the digraph G, in some unspecified order.

out_neighbours(G, V) → Vertices
Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Returns a list of all out-neighbours [page 80] of \( V \) of the digraph \( G \), in some unspecified order.

\[ \text{vertex}(G, V) \rightarrow \{ V, \text{Label} \} \mid \text{false} \]

Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex()} \)
- \( \text{Label} = \text{label()} \)

Returns \( \{ V, \text{Label} \} \) where \( \text{Label} \) is the label [page 80] of the vertex \( V \) of the digraph \( G \), or \text{false} if there is no vertex \( V \) of the digraph \( G \).

\[ \text{vertices}(G) \rightarrow \text{Vertices} \]

Types:
- \( G = \text{digraph}() \)
- \( \text{Vertices} = [\text{vertex()}] \)

Returns a list of all vertices of the digraph \( G \), in some unspecified order.

See Also

digraph_utils(3) [page 87], ets(3) [page 116]
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 \(V \times V\) (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 \(1 \leq i < k\). 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 87]. 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 87] 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 87] from x and incident [page 87] 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 87].

Each and every cycle [page 87] is included in some strongly connected component, which implies that there always exists a topological ordering [page 87] of the created digraph.

cyclic_strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components [page 87]. 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 87] 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 87].

loop_vertices(Digraph) -> Vertices

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

Returns a list of all vertices of Digraph that are included in some loop [page 87].
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 87] 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 87] 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 87] 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 87] 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 87] 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 87] 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 87] 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 87] are returned.

strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components [page 87]. Each strongly connected 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 strongly connected 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 87] 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 87] 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 87] 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 87] that occur earlier in the list.

See Also
digraph(3) [page 80]
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}
oopen(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()
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 102]
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}.
erl_eval

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:

\[
\text{Func}(\text{FuncSpec}, \text{Arguments})
\]

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

\texttt{parse_transform(Forms, Options)} \rightarrow \texttt{Forms}

Types:
- \texttt{Forms = [erlang_form()]}  
- \texttt{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.

\textbf{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 102], 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

\[
\text{module}(\text{AbsForms}) \rightarrow \{\text{ok},\text{Warnings}\} \mid \{\text{error},\text{Errors},\text{Warnings}\}
\]

\[
\text{module}(\text{AbsForms}, \text{FileName}) \rightarrow \{\text{ok},\text{Warnings}\} \mid \{\text{error},\text{Errors},\text{Warnings}\}
\]

\[
\text{module}(\text{AbsForms}, \text{FileName}, \text{CompileOptions}) \rightarrow \{\text{ok},\text{Warnings}\} \mid \{\text{error},\text{Errors},\text{Warnings}\}
\]

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:

\[
\{\text{ok},\text{Warnings}\} \text{ There were no errors in the module.}
\]

\[
\{\text{error},\text{Errors},\text{Warnings}\} \text{ There were errors in the module.}
\]
The elements of Options selecting optional warnings are as follows:

\{warn_format, Verbose\} Causes warnings to be emitted for malformed format strings as arguments to io:format and similar functions. Verbose 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. Verbose 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 91], or parser erl_parse(3) [page 102]. 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(ErrorCode) -> string()
Types:
- ErrorCode = erordesc()
Takes an ErrorCode 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 102], epp(3) [page 91]
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(Errordesc) -> 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 180], erl_scan(3) [page 108], 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 180], erl_parse(3) [page 102], erl_eval(3) [page 93]
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:
- \(\text{CharList} = \text{string()}\)
- \(\text{StartLine} = \text{EndLine} = \text{Line} = \text{integer()}\)
- \(\text{Tokens} = [\{$\text{atom()}$,\text{Line}\} | \{\$\text{atom()}$,\text{Line}$,\text{term()}$\}\]
- \(\text{Error} = \{\text{error}, \text{ErrorInfo}, \text{EndLine}\}\)

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:
- \(\text{Return} = \{\text{done}, \text{Result}, \text{LeftOverChars}\} \mid \{\text{more}, \text{Continuation}\}\)
- \(\text{Continuation} = [\] \mid \text{string()}\)
- \(\text{CharList} = \text{string()}\)
- \(\text{StartLine} = \text{EndLine} = \text{integer()}\)
- \(\text{Result} = \{\text{ok, Tokens, EndLine}\} \mid \{\text{eof, EndLine}\}\)
- \(\text{Tokens} = [\{$\text{atom()}$,\text{Line}\} | \{\$\text{atom()}$,\text{Line}$,\text{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 = erordesc()
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 180], erl_parse(3) [page 102]
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/IEC 9945-1. 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 112] or `create/3` [page 112] function. Alternatively, for more control, the `open` [page 114], `add/3,4` [page 111], and `close/1` [page 111] functions can be used.

To extract all files from a tar file, use the `extract/1` [page 112] function. To extract only some files or to be able to specify some more options, use the `extract/2` [page 113] function.

To return a list of the files in a tar file, use either the `table/1` [page 114] or `table/2` [page 114] function. To print a list of files to the Erlang shell, use either the `t/1` [page 114] or `tt/1` [page 115] function.

To convert an error term returned from one of the functions above to a readable message, use the `format_error/1` [page 113] 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 114].

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 114]. It accepts the same options as add/3 [page 111].

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 114].

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.

```erl
create(Name, FileList, OptionList)
```

Types:
- `Name = filename()`
- `FileList = [filename()]`
- `OptionList = [Option]`
- `Option = compressed | cooked | dereference | verbose`
- `RetVal = ok | {error, {Name, Reason}}`

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.

```erl
extract(Name) -> RetValue
```

Types:
- `Name = filename()`
- `RetVal = 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.

```erl
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()]`
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, FileList}` 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 111] functions to add one file at the time into an opened tar file. When you are finished adding files, use the `close` [page 111] 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.

```
\[table\]
```

### `table(Name)`

Returns:

- `ok, [string()] | error, Name, Reason`

Types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>filename()</td>
</tr>
</tbody>
</table>

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

### `table(Name, Options)`

Types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>filename()</td>
</tr>
</tbody>
</table>

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

### `t(Name)`

Types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>filename()</td>
</tr>
</tbody>
</table>

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

### `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 "tar tv").
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() \rightarrow [\text{Tab}]

Types:
- \text{Tab} = \text{tid()} | \text{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) \rightarrow \text{true}

Types:
- \text{Tab} = \text{tid()} | \text{atom()}

Deletes the entire table \text{Tab}.

delete(Tab, Key) \rightarrow \text{true}

Types:
- \text{Tab} = \text{tid()} | \text{atom()}
- \text{Key} = \text{term()}

Deletes all objects with the key \text{Key} from the table \text{Tab}.

delete\_all\_objects(Tab) \rightarrow \text{true}

Types:
- \text{Tab} = \text{tid()} | \text{atom()}

Delete all objects in the ETS table \text{Tab}. The deletion is atomic.

delete\_object(Tab, Object) \rightarrow \text{true}

Types:
- \text{Tab} = \text{tid()} | \text{atom()}
- \text{Object} = \text{tuple()}

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

file2tab(Filename) \rightarrow \{\text{ok,Tab} | \text{error,Reason}\}

Types:
- \text{Filename} = \text{string()} | \text{atom()}
- \text{Tab} = \text{tid()} | \text{atom()}
- \text{Reason} = \text{term()}

Reads a file produced by tab2file/2 and creates the corresponding table \text{Tab}.

first(Tab) \rightarrow \text{Key} \mid \text{'$end\_of\_table'}

Types:
- \text{Tab} = \text{tid()} | \text{atom()}
- \text{Key} = \text{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. Since safe_fixtable/2 is called, the table must be public or owned by the calling process.

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. Since safe_fixtable/2 is called, the table must be public or owned by the calling process.

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.

```plaintext
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").
```

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, receive 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 cannot 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}
```

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 being 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.

```erlang
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.

```erlang
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.

```erlang
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`.

```erlang
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:

```erlang
ets:match(T, 'rufsen','dog',7).
[[rufsen,dog,7]]
```

`ets:match(T, 'brunte','horse',5).
[[brunte,horse,5]]`

`ets:match(T, 'ludde','dog','$1').
[[ludde,dog,5]]`

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.

```erlang
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.

```erlang
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} Specifies 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:
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
degrad 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 described 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>` expression:

```erlang
ets:match(Tab, ['$1', '$2', '$3'])
```

is exactly equivalent to:

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

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

```erlang
ets:match_object(Tab, ['$1', '$2', '$1'])
```

is exactly equivalent to

```erlang
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:

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

gives the same output as:

```erlang
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 being the tuple one wants to construct (as an ordinary tuple could be mistaken for a guard). Therefore the following call:

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

gives the same output as:

```erlang
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:

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

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

```erlang
[[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.

```erlang
select(Tab, MatchSpec, Limit) -> [{Match}, Continuation] | '$end of table'
```
Types:
- Tab = tidy() | atom()
- Object = tuple()
- MatchSpec = term()
- Continuation = term()

Work 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 = tidy() | 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 = tidy() | atom()
- Object = tuple()
- MatchSpec = term()
- NumMatched = integer()
Matches the objects in the table \texttt{Tab} using a \texttt{match_spec} as described in ETS users guide. If the \texttt{match_spec} returns the atom \texttt{true} for an object, that object considered a match and is counted. For any other result from the \texttt{match_spec} the object is not considered a match and is therefore not counted.

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

The function returns the number of objects matched.

\begin{verbatim}
slot(Tab, I) -> [Object] | '$end_of_table'

Types:
- \texttt{Tab} = tid() \texttt{|} atom()
- \texttt{I} = int()
- \texttt{Object} = tuple()

This function is mostly for debugging purposes. Normally one should use \texttt{first/next} or \texttt{last/prev} instead.

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

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

\begin{verbatim}
tab2file(Tab, Filename) -> ok | {error,Reason}

Types:
- \texttt{Tab} = tid() \texttt{|} atom()
- \texttt{Filename} = string() \texttt{|} atom()
- \texttt{Reason} = term()

Dumps the table \texttt{Tab} to the file \texttt{Filename}. The implementation of this function is not efficient.
\end{verbatim}

\begin{verbatim}
tab2list(Tab) -> [Object]

Types:
- \texttt{Tab} = tid() \texttt{|} atom()
- \texttt{Object} = tuple()

Returns a list of all objects in the table \texttt{Tab}.
\end{verbatim}

\begin{verbatim}
test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}

Types:
- \texttt{Tuple} = tuple()
- \texttt{MatchSpec} = term()
- \texttt{Result} = term()
- \texttt{Errors} = [{\texttt{warning} \texttt{|} \texttt{error} \texttt{|} \texttt{string}()}]
\end{verbatim}
This function is a utility to test the `matchspec`'s used in calls to `ets:select/2`. The function both tests the `matchspec` for "syntactic" correctness and runs the `matchspec` against the object `Tuple`. If the `matchspec` contains errors, the tuple `{error, Errors}` is returned where `Errors` is a list of natural language descriptions of what was wrong with the `matchspec`. If the `matchspec` 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 `matchspec` 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

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 (\(\geq 0\)) 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 `fun binary_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 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.

```erlang
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.
```

```erlang
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.
```

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

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

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

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

Erlang Module

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

Exports

```erlang
ensure_dir(Name) -> true
```

Types:

- `Name = filename() | dirname()`

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

```erlang
file_size(Filename) -> integer()
```

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

```erlang
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.

```erlang
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.

```erlang
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/DO\S 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 can be used:
```
    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}"")
```
Types:
- \texttt{Wildcard} = \texttt{filename()} \mid \texttt{dirname()}
- \texttt{Cwd} = \texttt{dirname()}

The \texttt{ wildcard/2} function works like \texttt{ wildcard/1}, except that instead of the actual working directory, \texttt{Cwd} will be used.
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 names 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"
(forn UNIX): `absname("/") -> "/"
(forn WIN32): `absname("/") -> "D:/"

`absname(Filename, Directory) -> Absname`
Types:
- Filename = filename()
- Directory = string()
- Absname = string()

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

**basename(Filename)**

Types:
- Filename = 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 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("C:\\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:

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

**join(Components) -> string()**

Types:

- `Components = [string()]`

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

The result of the `join` function is “normalized”:

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

Examples include:

```erlang
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])`.

**nativeness(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:nativeness("/usr/local/bin")` -> "/usr/local/bin"
- (on Win32) `filename:nativeness("/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 \( O(|T| \times \log(|S|)) \), 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
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 \( X, S_1 \), where X is the smallest element in set S, and \( S_1 \) is the set S with element X deleted. Assumes that the set S is nonempty.

take_largest(S)
Returns \( X, S_1 \), where X is the largest element in set S, and \( S_1 \) 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 \( X, T_1 \) where X is the smallest element referred to by the iterator T, and \( T_1 \) 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 149], ordsets(3) [page 220], sets(3) [page 244]
gb_trees

Erlang Module

An efficient implementation of Prof. Arne Andersson’s General Balanced Trees. These have no storage overhead compared to unbalanced 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:
  - \text{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 \( h(T) \leq \text{ceil}(c \times \log(|T|)) \) has been changed to the similar (but not quite equivalent) condition \( 2^{-h(T)} \leq |T| \leq c \). 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 logarithmic (as it should be).

Exports

\texttt{empty()}

Returns a new, empty tree.

\texttt{is_empty(T)}

Returns ‘true’ if \( T \) is an empty tree, and ‘false’ otherwise.

\texttt{size(T)}

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

\texttt{lookup(X, T)}

Looks up key \( X \) in tree \( T \); returns \{value, V\}, or ‘none’ if the key is not present.

\texttt{get(X, T)}
Retrieves the value stored with key X in tree T. Assumes that the key is present in the tree, crashes otherwise.

\textbf{insert}(X, V, T)\par
Inserts key X with value V into tree T; returns the new tree. Assumes that the key is \textit{not} present in the tree, crashes otherwise.

\textbf{update}(X, V, T)\par
Updates key X to value V in tree T; returns the new tree. Assumes that the key is present in the tree.

\textbf{enter}(X, V, T)\par
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.

\textbf{delete}(X, T)\par
Removes key X from tree T; returns new tree. Assumes that the key is present in the tree, crashes otherwise.

\textbf{delete\_any}(X, T)\par
Removes key X from tree T if the key is present in the tree, otherwise does nothing; returns new tree.

\textbf{balance}(T)\par
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.

\textbf{is\_defined}(X, T)\par
Returns 'true' if key X is present in tree T, and 'false' otherwise.

\textbf{keys}(T)\par
Returns an ordered list of all keys in tree T.

\textbf{values}(T)\par
Returns the list of values for all keys in tree T, sorted by their corresponding keys. Duplicates are not removed.

\textbf{to\_list}(T)\par
Returns an ordered list of \{(Key, Value)\} pairs for all keys in tree T.
Trees

Turns an ordered list \( L \) of \((\text{Key}, \text{Value})\) pairs into a tree. The list must not contain duplicate keys.

\texttt{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.

\texttt{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.

\texttt{take\_smallest(T)}

Returns \((X, V, T_1)\), where \( X \) is the smallest key in tree \( T \), \( V \) is the value associated with \( X \) in \( T \), and \( T_1 \) is the tree \( T \) with key \( X \) deleted. Assumes that the tree \( T \) is nonempty.

\texttt{take\_largest(T)}

Returns \((X, V, T_1)\), where \( X \) is the largest key in tree \( T \), \( V \) is the value associated with \( X \) in \( T \), and \( T_1 \) is the tree \( T \) with key \( X \) deleted. Assumes that the tree \( T \) is nonempty.

\texttt{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.

\texttt{next(S)}

Returns \((X, V, S_1)\) where \( X \) is the smallest key referred to by the iterator \( S \), and \( S_1 \) is the new iterator to be used for traversing the remaining entries, or the atom ‘none’ if no entries remain.

\textbf{SEE ALSO}

\texttt{gb\_sets(3) \[page 145\], dict(3) \[page 76\],}
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:

```
<table>
<thead>
<tr>
<th>gen_event module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_event:start</td>
<td></td>
</tr>
<tr>
<td>gen_event:add_handler</td>
<td>Module:init/1</td>
</tr>
<tr>
<td>gen_event:add_suphandler</td>
<td>Module: init/2</td>
</tr>
<tr>
<td>gen_event:notify</td>
<td>Module:handle_event/2</td>
</tr>
<tr>
<td>gen_event:call</td>
<td>Module:handle_call/2</td>
</tr>
<tr>
<td>-</td>
<td>Module:handle_info/2</td>
</tr>
<tr>
<td>gen_event:delete_handler</td>
<td>Module:terminate/2</td>
</tr>
<tr>
<td>gen_event:swap_handler</td>
<td>Module1: terminate/2</td>
</tr>
<tr>
<td>gen_event:swap_sup_handler</td>
<td>Module2: init/1</td>
</tr>
<tr>
<td>gen_event:which_handlers</td>
<td></td>
</tr>
<tr>
<td>gen_event:stop</td>
<td>Module: terminate/2</td>
</tr>
<tr>
<td>-</td>
<td>Module:code_change/3</td>
</tr>
</tbody>
</table>
```

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

\begin{verbatim}
start() -> Result
start(EventMgrName) -> Result
start_link() -> Result
start_link(EventMgrName) -> Result
\end{verbatim}

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.

\begin{verbatim}
add_handler(EventMgrRef, Handler, Args) -> Result
\end{verbatim}

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:
• EventM gr = 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
STDLIB Reference Manual

event

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 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 287], sys(3) [page 297]
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:

```
<table>
<thead>
<tr>
<th>gen_fsm module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_fsm:start_link</td>
<td>------ &gt; Module:init/1</td>
</tr>
<tr>
<td>gen_fsm:send_event</td>
<td>------ &gt; Module:StateName/2</td>
</tr>
<tr>
<td>gen_fsm:send_all_state_event</td>
<td>------ &gt; Module:handle_event/3</td>
</tr>
<tr>
<td>gen_fsm:sync_send_event</td>
<td>------ &gt; Module:StateName/3</td>
</tr>
<tr>
<td>gen_fsm:sync_send_all_state_event</td>
<td>------ &gt; Module:handle_sync_event/4</td>
</tr>
<tr>
<td>-</td>
<td>------ &gt; Module:handle_info/3</td>
</tr>
<tr>
<td>-</td>
<td>------ &gt; Module:terminate/3</td>
</tr>
<tr>
<td>-</td>
<td>------ &gt; Module:code_change/4</td>
</tr>
</tbody>
</table>
```

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`
- `Time` = `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) \rightarrow \text{ok}

Types:

- \text{FsmRef} = \text{Name} | \{\text{Name},\text{Node}\} | \{\text{global},\text{Name}\} | \text{pid()}
- \text{Name} = \text{Node} = \text{atom()}
- \text{Event} = \text{term()}

Sends an event asynchronously to the gen\_fsm FsmRef and returns \text{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) \rightarrow \text{ok}

Types:

- \text{FsmRef} = \text{Name} | \{\text{Name},\text{Node}\} | \{\text{global},\text{Name}\} | \text{pid()}
- \text{Name} = \text{Node} = \text{atom()}
- \text{Event} = \text{term()}

Sends an event asynchronously to the gen\_fsm FsmRef and returns \text{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 \text{send\_event} and \text{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) \rightarrow \text{Reply}

sync\_send\_event(FsmRef, Event, Timeout) \rightarrow \text{Reply}

Types:

- \text{FsmRef} = \text{Name} | \{\text{Name},\text{Node}\} | \{\text{global},\text{Name}\} | \text{pid()}
- \text{Name} = \text{Node} = \text{atomic()}
- \text{Event} = \text{term()}
- \text{Timeout} = \text{int()} > 0 | \text{infinity}
- \text{Reply} = \text{term()}

STDLIB
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.

\[
\text{sync_send_all_state_event}(\text{FsmRef}, \text{Event}) \rightarrow \text{Reply}
\]
\[
\text{sync_send_all_state_event}(\text{FsmRef}, \text{Event}, \text{Timeout}) \rightarrow \text{Reply}
\]

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.

\[
\text{reply}(\text{Caller}, \text{Reply}) \rightarrow \text{true}
\]

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.

\[
\text{send_event_after}(\text{Time}, \text{Event}) \rightarrow \text{Ref}
\]

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,Timeout}, where StateName is the initial state name and StateData the initial state data of the gen_fsm.

If an integer timeout value is provided, a timeout will occur unless an event or a message is received within Timeout milliseconds. A timeout is represented by the atom timeout and should be handled by the Module:StateName/2 callback functions. The atom infinity 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 occurred, 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:syncc_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()
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.

```erlang
Module:terminate(Reason, StateName, StateData) Type:
  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 supervisor 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`.

```erlang
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 287], sys(3) [page 297]
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:

```
<table>
<thead>
<tr>
<th>gen_server module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_server:start</td>
<td>Module:init/1</td>
</tr>
<tr>
<td>gen_server:call</td>
<td>Module:handle_call/3</td>
</tr>
<tr>
<td>gen_server:multi_call</td>
<td>Module:handle_cast/2</td>
</tr>
<tr>
<td>gen_server:cast</td>
<td>Module:handle_cast/2</td>
</tr>
<tr>
<td>gen_server:abcast</td>
<td>Module:handle_info/2</td>
</tr>
<tr>
<td></td>
<td>Module:terminate/2</td>
</tr>
<tr>
<td></td>
<td>Module:code_change/3</td>
</tr>
</tbody>
</table>
```

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,Name} | pid()
- Name = Node = atom()
- Request = term()
- Timeout = int()>=0 | infinity
- \( \text{Reply} = \text{term()} \)

Makes a synchronous call to the \texttt{gen_server} \texttt{ServerRef} by sending a request and waiting until a reply arrives or a timeout occurs. The \texttt{gen_server} will call \texttt{Module:handle_call/3} to handle the request.

\texttt{ServerRef} can be:

- the \texttt{pid},
- \texttt{Name}, if the \texttt{gen_server} is locally registered,
- \{\texttt{Name},\texttt{Node}\}, if the \texttt{gen_server} is locally registered at another node, or
- \{\texttt{global},\texttt{Name}\}, if the \texttt{gen_server} is globally registered.

\texttt{Request} is an arbitrary term which is passed as one of the arguments to \texttt{Module:handle_call/3}.

\texttt{Timeout} is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom \texttt{infinity} to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value \texttt{Reply} is defined in the return value of \texttt{Module:handle_call/3}.

The call may fail for several reasons, including timeout and the called \texttt{gen_server} dying before or during the call.

There is a special case for backwards compatibility. If

- the client is linked to the \texttt{gen_server}, and
- the client is trapping exits, and
- the \texttt{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 \texttt{gen_server} dies between calls and also because the exit message cannot be removed in some circumstances, for instance when \texttt{ServerRef} = \{\texttt{Name}, \texttt{Node}\} and \texttt{Node} goes down.

\begin{verbatim}
multi_call(Name, Request) -> Result
multi_call(Nodes, Name, Request) -> Result
multi_call(Nodes, Name, Request, Timeout) -> Result
\end{verbatim}

Types:

- \texttt{Nodes} = \texttt{[Node]}  
- \texttt{Node} = \texttt{atom()}  
- \texttt{Name} = \texttt{atom()}  
- \texttt{Request} = \texttt{term()}  
- \texttt{Timeout} = \texttt{int()} \geq 0 | \texttt{infinity}  
- \texttt{Result} = \{\texttt{Replies,BadNodes}\}  
- \texttt{Replies} = \{\texttt{[Node,Reply]}\}  
- \texttt{Reply} = \texttt{term()}  
- \texttt{BadNodes} = \texttt{[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.

\texttt{cast(ServerRef, Request) -\rightarrow ok}

Types:
- \texttt{ServerRef = Name \| \{Name, Node\} \| \{global, Name\} \| pid()}
- \texttt{Name = Node = atom()}
- \texttt{Request = term()}

Sends an asynchronous request to the gen_server ServerRef and returns ok immediately. 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.

\texttt{abcast(Name, Request) -\rightarrow abcast}

\texttt{abcast(Nodes, Name, Request) -\rightarrow abcast}
Types:
- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()

Sends an asynchronous request to the gen_server 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,Name}
- Name = atom()
- 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 initialization 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,Timeout}, where State is the internal state of the gen_server.

If an integer timeout value is provided, a timeout will occur unless a request or a message is received within Timeout milliseconds. A timeout is represented by the atom timeout which should be handled by the handle_info/2 callback function. The atom infinity 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,Timeout}`, 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` or `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 occurred, 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,\ldots\}, 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 287], sys(3) [page 297]
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 \texttt{ok} if successful 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.

```erl
write([IoDevice,] Term)
\text{Writes the term \texttt{Term} to the standard output (IoDevice).}
```

```erl
read([IoDevice,] Prompt)
\text{Reads a term from the standard input (IoDevice), prompting it with \texttt{Prompt}. It returns:}
\texttt{ok, Term} The parsing was successful.
\texttt{error, ErrorInfo} The parsing failed.
\texttt{eof} End of file was encountered.
```

```erl
fwrite(Format)
format(Format)
\text{Equivalent to \texttt{fwrite(Format, [])}.}
```

```erl
fwrite([IoDevice,] Format, Arguments)
format([IoDevice,] Format, Arguments)
\text{Writes the list of items in \texttt{Arguments} on the standard output (IoDevice) in accordance with \texttt{Format}. \texttt{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 \texttt{Format} is an atom, it is first converted to a list with the aid of \texttt{atom_to_list/1}. \texttt{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.P\text{Pad}C. 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:

\begin{verbatim}
> io:fwrite("|~10.5c|~-10.5c|~5c|~n", [$a, $b, $c]).
| aaaaa|aaaaa |ccccc|
ok
\end{verbatim}

\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:

\begin{verbatim}
> io:fwrite("|~10w|~n", [{hey, hey, hey}]).
|**********|
ok
> io:fwrite("|~10s|~n", [io_lib:write({hey, hey, hey})]).
|{hey, hey, h|
ok
\end{verbatim}

\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 \texttt{~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:

\begin{verbatim}
> 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}, {typename, '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
\end{verbatim}

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 \texttt{io:fwrite} or \texttt{io:format}. For example,
using \(T\) above:

\begin{verbatim}
> 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
\end{verbatim}

w  Writes data in the same way as \texttt{~w}, but takes an extra argument which is the
maximum depth to which terms are printed. Anything below this depth is
replaced with \ldots. For example, using \(T\) above:

\begin{verbatim}
> io:fwrite("~W~n", [T, 9]).

[{attributes, [{id, age, 1.50000}, {mode, explicit},
  {typename, ...
  ...}], [{id, cho}, {mode, ...
  ...}], {typename, 'Person'}, {tag, {'PRIVATE', 3}}, {mode, implicit}]

ok
\end{verbatim}

If the maximum depth has been reached, then it is impossible to read in the
resultant output. Also, the |\ldots 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,1.500000},{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:

```erlang
> 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.
  - A decimal integer is expected.
  - 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.
  - A floating point number is expected. It must follow the Erlang floating point number syntax.
  - 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.
  - Similar to a, but the resulting string is converted into an atom.
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.

1 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:

```erl
good> 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)

```erl
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:

```erl
> 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)

```erl
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>').' abc("hey").

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:
    ```erlang```
    > 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 180] 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 180] 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)\nThis 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)\nReturns the list of characters needed to print the atom Atom.

\write_string(String)\nReturns the list of characters needed to print String as a string.

\write_char(Integer)\nReturns the list of characters needed to print a character constant.

\indentation(String, StartIndent)\nReturns the indentation if String has been printed, starting at Indentation.

\char_list(CharList) \(\rightarrow\) bool()
Returns true if CharList is a list of characters, otherwise it returns false.

\texttt{deep\_char\_list(CharList)}

Returns true if CharList is a deep list of characters, otherwise it returns false.

\texttt{printable\_list(CharList)}

Returns true if CharList is a list of printable characters, otherwise it returns false.

Notes

The module \texttt{io\_lib} also uses the extra modules \texttt{io\_lib\_format}, \texttt{io\_lib\_fread}, and \texttt{io\_lib\_pretty}. All external interfaces exist in \texttt{io\_lib}.

Users are strongly advised not to access the other modules directly.

\textbf{Note:}

Any undocumented functions in \texttt{io\_lib} should not be used.

The continuation of the first call to the re-entrant input functions must be 
\texttt{[]}{}. 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.

erro_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 improper 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 \(N\)th element is Key is deleted, if present.

keymember(Key, N, TupleList) -> bool()

Types:
- TupleList = [tuple()]
- N = int()
- Key = term()

Searches the list of tuples TupleList for a tuple whose \(N\)th element is Key.

keymerge(N, List1, List2)

Types:
- N = int()
- List1 = List2 = [tuple()]

Returns the sorted list formed by merging List1 and List2. The merge is performed on the \(N\)th element of each tuple. Both List1 and List2 must be key-sorted prior to evaluating this function; otherwise the 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 \(N\)th element is Key. Returns \{(value, Tuple)\} if such a tuple is found, or false if no such tuple is found.

keysort(N, List1) -> List2
Types:
- N = int()
- List1 = List2 = [tuple()]

Returns a list containing the sorted elements of List1. TupleList1 must be a list of tuples, and the sort is performed on the Nth 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 \texttt{List1} and \texttt{List2}. Both \texttt{List1} and \texttt{List2} must be sorted according to the ordering function \texttt{Fun} prior to evaluating this function. \texttt{Fun(A,B)} should return \texttt{true} if \texttt{A} comes before \texttt{B} in the ordering, \texttt{false} otherwise.

\texttt{merge3(List1, List2, List3) \rightarrow List4}

Types:
- \texttt{List1 = List2 = List3 = List4 = \{}term\(\})

Returns the sorted list formed by merging \texttt{List1}, \texttt{List2} and \texttt{List3}. All of \texttt{List1}, \texttt{List2} and \texttt{List3} must be sorted prior to evaluating this function.

\texttt{min(List) \rightarrow Min}

Types:
- \texttt{List = \{}Element\(\)}
- \texttt{Element = Max = term()}

Returns the minimum element of \texttt{List}.

\texttt{nth(N, List) \rightarrow Element}

Types:
- \texttt{N = int()}
- \texttt{List = \{}Element\(\)}
- \texttt{Element = term()}

Returns the \texttt{N}th element of the \texttt{List}. For example:

\texttt{> lists:nth(3, \{}a, b, c, d, e\}\}).
\texttt{c}

\texttt{nthtail(N, List1) \rightarrow List2}

Types:
- \texttt{N = int()}
- \texttt{List1 = List2 = \{}Alpha\(\)}

Returns the \texttt{N}th tail of \texttt{List}. For example:

\texttt{> lists:nthtail(3, \{}a, b, c, d, e\}\}).
\texttt{[d, e]}

\texttt{prefix(List1, List2) \rightarrow bool()}

Types:
- \texttt{List1 = List2 = \{}term\(\)}

Returns \texttt{true} if \texttt{List1} is a prefix of \texttt{List2}, otherwise \texttt{false}.

\texttt{reverse(List1) \rightarrow List2}

Types:
- \texttt{List1 = List2 = \{}term\(\)}

Returns a list with the top level elements in \texttt{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 Nth 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:
**STDLIB Reference Manual**

- $N = \text{int()}$
- $\text{List1} = \text{List2} = [\text{tuple()}]$

Returns a list containing the sorted elements of $\text{List1}$ with consecutive duplicates removed. $\text{TupleList1}$ must be a list of tuples, and the sort is performed on the $i$th element of the tuple. The sort is stable.

\[ \text{umerge}(\text{ListOfLists}) \rightarrow \text{List1} \]

Types:
- $\text{ListOfLists} = [\text{List}]$
- $\text{List} = \text{List1} = [\text{term()}]$

Returns the sorted list formed by merging all the sub-lists of $\text{ListOfLists}$ while removing duplicates. All sub-lists must be sorted and contain no duplicates prior to evaluating this function.

\[ \text{umerge} (\text{List1}, \text{List2}) \rightarrow \text{List3} \]

Types:
- $\text{List1} = \text{List2} = \text{List3} = [\text{term()}]$

Returns the sorted list formed by merging $\text{List1}$ and $\text{List2}$ while removing duplicates. Both $\text{List1}$ and $\text{List2}$ must be sorted and contain no duplicates prior to evaluating this function.

\[ \text{umerge} (\text{Fun}, \text{List1}, \text{List2}) \rightarrow \text{List} \]

Types:
- $\text{List} = \text{List1} = \text{List2} = [\text{Element}]$
- $\text{Fun} = \text{fun(} \text{Element, Element} \text{)} \rightarrow \text{bool()}$
- $\text{Element} = \text{term()}$

Returns the sorted list formed by merging $\text{List1}$ and $\text{List2}$ while removing consecutive duplicates. Both $\text{List1}$ and $\text{List2}$ must be sorted according to the ordering function $\text{Fun}$ prior to evaluating this function. $\text{Fun}(A,B)$ should return true if $A$ comes before $B$ in the ordering, false otherwise.

\[ \text{umerge3} (\text{List1}, \text{List2}, \text{List3}) \rightarrow \text{List4} \]

Types:
- $\text{List1} = \text{List2} = \text{List3} = \text{List4} = [\text{term()}]$

Returns the sorted list formed by merging $\text{List1}$, $\text{List2}$ and $\text{List3}$ while removing duplicates. All of $\text{List1}$, $\text{List2}$ and $\text{List3}$ must be sorted and contain no duplicates prior to evaluating this function.

\[ \text{usort} (\text{List1}) \rightarrow \text{List2} \]

Types:
- $\text{List1} = \text{List2} = [\text{term()}]$

Returns a list which contains the sorted elements of $\text{List1}$ without duplicates.

\[ \text{usort} (\text{Fun}, \text{List1}) \rightarrow \text{List2} \]
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.

\[ \text{all}(\text{Pred}, \text{List}) \rightarrow \text{bool}() \]

Types:
- Pred = fun(A) -> bool()
- List = [A]

Returns true if all elements X in List satisfy Pred(X).

\[ \text{any}(\text{Pred}, \text{List}) \rightarrow \text{bool}() \]

Types:
- Pred = fun(Element) -> bool()
- List = [Element]
- Element = term()

Returns true if any of the elements in List satisfies Pred.

\[ \text{dropwhile}(\text{Pred}, \text{List1}) \rightarrow \text{List2} \]

Types:
- Pred = fun(A) -> bool()
- List1 = List2 = [A]

Drops elements X from List1 while Pred(X) is true and returns the remaining list.

\[ \text{filter}(\text{Pred}, \text{List1}) \rightarrow \text{List2} \]

Types:
- Pred = fun(A) -> bool()
- List1 = List2 = [A]

List2 is a list of all elements X in List1 for which Pred(X) is true.

\[ \text{flatmap}(\text{Function}, \text{List1}) \rightarrow \text{Element} \]

Types:
- Function = fun(A) -> B
- List1 = [A]
- Element = [B]

flatmap behaves as if it had been defined as follows:

\[ \text{flatmap}(\text{Func}, \text{List}) \rightarrow \]
\[ \quad \text{append}(\text{map}(\text{Func}, \text{List})) \]

\[ \text{foldl}(\text{Function}, \text{Acc0}, \text{List}) \rightarrow \text{Acc1} \]
Types:
- Function = fun(A, AccIn) -> AccOut
- List = [A]
- Acc0 = Acc1 = AccIn = AccOut = term()

Acc0 is returned if the list is empty. For example:

```erlang
> 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
```

call 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.

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.

```erlang
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.

```erlang
map(Func, List1) -> List2
```

Types:
- Func = fun(A) -> B
- List1 = [A]
- List2 = [B]

map takes a function from As to Bs, and a list of As and produces a list of Bs by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.

```erlang
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:

```erlang
> 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 152], rb(3)
This module provides an interface to a number of mathematical functions.

Exports

\( \text{pi()} \) -> float()

A useful number.

\( \text{sin}(X) \)
\( \text{cos}(X) \)
\( \text{tan}(X) \)
\( \text{asin}(X) \)
\( \text{acos}(X) \)
\( \text{atan}(X) \)
\( \text{atan2}(Y, X) \)
\( \text{sinh}(X) \)
\( \text{cosh}(X) \)
\( \text{tanh}(X) \)
\( \text{asinh}(X) \)
\( \text{acosh}(X) \)
\( \text{atanh}(X) \)
\( \text{exp}(X) \)
\( \text{log}(X) \)
\( \text{log10}(X) \)
\( \text{pow}(X, Y) \)
\( \text{sqrt}(X) \)

Types:
- \( X = Y = \text{number}() \)
A collection of math functions which return floats. Arguments are numbers.

\( \text{erf}(X) \) -> float()

Types:
- \( X = \text{number}() \)
Returns the error function of \( X \), where

\( \text{erf}(X) = \frac{2}{\sqrt{\pi}} \int_{0}^{X} e^{-t^2} \, dt. \)
erfc(X) -> float()

Types:
- X = number()

erfc(X) returns \(1.0 - \text{erf}(X)\), 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: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 straightforward 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:

```erlang
-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:

```erlang
ets:new(emp_tab,[{keypos,#emp.empno},named_table,ordered_set]).
```

```erlang
-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
```
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, "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 too small to justify complicated ets searches, but on real tables, using ets:select to get exactly the data you want will increase efficiency remarkably.

Let's 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:

```erlang
-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 when using the `ets:fun2ms` example, as the call to 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):

```erlang
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:

```erlang
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 in the guard in the `foldr` example. Let’s 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};
    end)).
(#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_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}]}

214
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):

```erlang
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:

```erlang
(<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 esoteric 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).
  \[{{'$1', '$2'}, \{>'$1', '$2', \{const, 25\}\}, \{'$2'\}}\]
  ```

- 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).
  \[\{\{'$1', \{>'$1', '$2'\}\}, \{>'$1', '$2', \{const, 25\}\}, \{>'$1', '$2', \{>'$_'\}\}\}\]
  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]} = D) 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: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 as BoundEnvironment. The result is a term, normalized, i.e. not in abstract format.

format_error(Errcode) -> ErrorMessage

Types:
- Errcode = term()
- ErrorMessage = 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 76], gb_trees(3) [page 149]
Sets are collections of elements with no duplicate elements. An order set 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 145], sets(3) [page 244]
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 empty 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.
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**

```erlang
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`.

```erlang
spawn_link(Fun) -> Pid
spawn_link(Node,Fun) -> Pid
spawn_link(Module,Func,Args) -> Pid
spawn_link(Node,Module,Func,Args) -> Pid
```

**Types:**

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.

**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.

\[ \text{initial\_call(PidOrPinfo)} \rightarrow \{\text{Module, Function, Args} \} \text{ | Fun \ | false} \]

Types:
- \( \text{PidOrPinfo = \{\text{pid() | \{X,Y,Z} | \text{ProcInfo}} \}
- \( X = Y = Z = \text{int()} \)
- \( \text{ProcInfo = \{\text{void()}} \)
- \( \text{Module = \text{atom()} \)
- \( \text{Fun = fun() -> void() \)
- \( \text{Function = \text{atom()} \)
- \( \text{Args = \{\text{term()}} \)

Extracts the initial call of a process that was spawned using the spawn functions described above. \( \text{PidOrPinfo} \) 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 \text{erlang:process_info/1} function call).

\[ \text{translate\_initial\_call(PidOrPinfo) \rightarrow \{\text{Module, Function, Arity} \} \text{ | Fun} \]

Types:
- \( \text{PidOrPinfo = \{\text{pid() | \{X,Y,Z} | \text{ProcInfo}} \)
- \( X = Y = Z = \text{int()} \)
- \( \text{ProcInfo = \{\text{void()}} \)
- \( \text{Module = \text{atom()} \)
- \( \text{Fun = fun() -> void() \)
- \( \text{Function = \text{atom()} \)
- \( \text{Arity = \text{int()}} \)

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 \text{gen_server} or \text{gen_event}, it is translated to more useful information. If a \text{gen_server} is spawned, the returned \text{Module} is the name of the callback module and \text{Function} is \text{init} (the function that initiates the new server).

A supervisor and a supervisor bridge are also \text{gen_server} processes. In order to return information that this process is a supervisor and the name of the callback module, \text{Module} is \text{supervisor} and \text{Function} is the name of the supervisor callback module. \text{Arity} is \text{1} since the \text{init/1} function is called initially in the callback module.

By default, \{\text{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 \text{proc\_lib} module.

\( \text{PidOrPinfo} \) 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 \text{erlang:process\_info/1} function call).

This function is used by the \text{c:i/0} and \text{c:regs/0} functions in order to present process information.

See Also

\text{error\_logger(3)}
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 occurrence 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.

```prolog
def_value(Key, List) -> term()

Types:
- Key = term()
- List = [term()]

Equivalent to get_value(Key, List, undefined).
```

```prolog
def_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.
```

```prolog
d_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.
```

```prolog
d_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.
```

```prolog
d_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.
```

```prolog
d_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 `substituteAliases/2` is applied using the given list of aliases; for a **negations** operation, `substituteNegations/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`, `substituteAliases/2`, `substituteNegations/2`.

### property(Preference) -> Property

**Types:**
- Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If `Preference` is `{Key, true}` where `Key` is an atom, this returns `Key`, otherwise the whole term `Preference` 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, true\} 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 occurrences 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 \texttt{len/1}, \texttt{reverse/1}, \texttt{join/2} and \texttt{split/2} that probably are $O(n)$.

Exports

\texttt{cons(Item, Q1) -> Q2}

Types:
\begin{itemize}
  \item Item = \texttt{term()}
  \item Q1 = Q2 = \texttt{queue()}
\end{itemize}

Inserts \texttt{Item} at the head of queue \texttt{Q1}. Returns the new queue \texttt{Q2}.

\texttt{daeh(Q) -> Item}

The same as \texttt{last(Q)} and the opposite of \texttt{head(Q)}.

\texttt{from_list(L) -> queue()}

Types:
\begin{itemize}
  \item L = \texttt{list()}
\end{itemize}

Returns a queue containing the items in \texttt{L}, in the same order - the head item of the list will be the head item of the queue.

\texttt{head(Q) -> Item}

Types:
\begin{itemize}
  \item Item = \texttt{term()}
  \item Q = \texttt{queue()}
\end{itemize}

Returns \texttt{Item} from the head of queue \texttt{Q}. Fails with reason \texttt{empty} if \texttt{Q} is empty.

\texttt{in(Item, Q1) -> Q2}

Types:
\begin{itemize}
  \item Item = \texttt{term()}
  \item Q1 = Q2 = \texttt{queue()}
\end{itemize}
Inserts Item at the tail of queue Q1. Returns a new queue Q2. This is the same as snoc(Q1, Item).

\[ \text{in}_r(\text{Item}, \text{Q1}) \rightarrow \text{Q2} \]

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).

\[ \text{init}(\text{Q1}) \rightarrow \text{Q2} \]

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.

\[ \text{is_empty}(\text{Q}) \rightarrow \text{true | false} \]

Types:
- Q = queue()

Tests if Q is empty and returns true if so and false otherwise.

\[ \text{join}(\text{Q1}, \text{Q2}) \rightarrow \text{Q3} \]

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.

\[ \text{lait}(\text{Q1}) \rightarrow \text{Q2} \]

The same as init(Q1) and the opposite of tail(Q1).

\[ \text{last}(\text{Q}) \rightarrow \text{Item} \]

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.

\[ \text{len}(\text{Q}) \rightarrow \text{N} \]

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
queue

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


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 \text{ran}()) consists of a tuple of three integers.

Exports

\text{seed()} \to \text{ran()}

Seeds random number generation with default (fixed) values in the process dictionary, and returns the old state.

\text{seed(A1, A2, A3)} \to \text{ran()}

Types:
\begin{itemize}
  \item \text{A1 = A2 = A3 = int()}
\end{itemize}

Seeds random number generation with integer values in the process dictionary, and returns the old state.

\text{seed0()} \to \text{ran()}

Returns the default state.

\text{uniform()} \to \text{float()}

Returns a random float uniformly distributed between 0.0 and 1.0, updating the state in the process dictionary.

\text{uniform(N)} \to \text{int()}

Types:
\begin{itemize}
  \item \text{N = int()}
\end{itemize}

Given an integer \text{N} \geq 1, \text{uniform/1} returns a random integer uniformly distributed between 1 and \text{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 $N \geq 1$ 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 = 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. A \& 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 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 \( \text{Set1} \) which are not also elements of \( \text{Set2} \).

\[
is\_\text{subset}(\text{Set1}, \text{Set2}) \rightarrow \text{bool()}
\]

Types:
- \( \text{Set1} = \text{Set2} = \text{set()} \)

Returns true when every element of \( \text{Set1} \) is also a member of \( \text{Set2} \), otherwise false.

\[
fold(\text{Function}, \text{Acc0}, \text{Set}) \rightarrow \text{Acc1}
\]

Types:
- \( \text{Function} = \text{fun (E, AccIn)} \rightarrow \text{AccOut} \)
- \( \text{Acc0} = \text{Acc1} = \text{AccIn} = \text{AccOut} = \text{term()} \)
- \( \text{Set} = \text{set()} \)

Fold \( \text{Function} \) over every element in \( \text{Set} \) returning the final value of the accumulator.

\[
filter(\text{Pred}, \text{Set1}) \rightarrow \text{Set2}
\]

Types:
- \( \text{Pred} = \text{fun (E)} \rightarrow \text{bool()} \)
- \( \text{Set1} = \text{Set2} = \text{set()} \)

Filter elements in \( \text{Set1} \) with boolean function \( \text{Fun} \).

See Also

ordsets(3) [page 220], gb_sets(3) [page 145]
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 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 Nth 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 shellDefault: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 ^Q)
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
").
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 \(\text{ok}\) 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 \((L, \text{Str}, \text{etc.})\) 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 \(\text{test1:demo(X)}\) 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.

`^G` (Control G) detaches the current job and JCL mode is activated. The JCL mode prompt is "--->". If "?" is entered at the prompt, the following help message is displayed:

```
---> ?
c [nn] - connect to job
i [nn] - interrupt job
k [nn] - kill job
j - list all jobs
s - start local shell
r [node] - start remote shell
q - quit Erlang
? | h - this message
```

The JCL commands have the following meaning:

c [nn] Connects to job number <nn> or the current job. The standard shell is resumed. Operations which use standard IO by the current job will be interleaved with user inputs to the shell.

i [nn] Stops the current evaluator process for job number nn or the current job, but does not kill the shell process. Accordingly, any variable bindings and the process dictionary will be preserved and the job can be connected again. This command can be used to interrupt an endless loop.

k [nn] Kills job number nn or the current job. All spawned processes in the job are killed, provided they have not evaluated the `group_leader/1` BIF and are located on the local machine. Processes spawned on remote nodes will not be killed.

j Lists all jobs. A list of all known jobs is printed. The current job name is prefixed with "*".

s Starts a new job. This will be assigned the new index [nn] which can be used in references.

r [node] Starts a remote job on node. This is used in distributed Erlang to allow a shell running on one node to control a number of applications running on a network of nodes.

q Quits Erlang.

? Displays this message.
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:

```erlang
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.

```erlang
non_local_allowed(FuncSpec, ArgList, State) -> {true,NewState} | {false,NewState}
```

to determine if the call to non-local function `FuncSpec` (either `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` 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`):
  ```erlang
  $ 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(Module) -> 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:loadabs("$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 \texttt{rsh}. The user must be allowed to \texttt{rsh} to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the \texttt{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 \texttt{rsh} program.

An alternative to the \texttt{rsh} program can be specified on the command line to \texttt{erl} as follows: \texttt{--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

\texttt{start(Host)}

Starts a slave node on the host \texttt{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 \texttt{start/3}.

\texttt{start_link(Host)}

Starts a slave node on the host \texttt{Host} in the same way as the \texttt{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 \texttt{start/3}.

\texttt{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)
slave

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`. 
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 $B = \{ x \in A : S(x) \}$. 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 elements 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 $X \times Y$, is the set $\{ a : a = (x, y) \text{ for some } x \in X \text{ and for some } y \in Y \}$. A relation is a subset of $X \times Y$. Let $R$ be a relation. The fact that $(x, y)$ belongs to $R$ is written as $x R y$. 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 $\{ x : x R y \text{ for some } y \in Y \}$. The range of $R$ is the set $\{ y : x R y \text{ for some } x \in X \}$. The converse of $R$ is the set $\{ a : a = (y, x) \text{ for some } (x, y) \in R \}$. If $A$ is a subset of $X$, then the image of $A$ under $R$ is the set $\{ y : x R y \text{ for some } x \in A \}$, and if $B$ is a subset of $Y$, then the inverse image of $B$ is the set $\{ x : x R y \text{ for some } y \in B \}$. 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 $x T z$ if and only if there exists an element $y$ in $Y$ such that $x R y$ and $y S z$. The restriction of $R$ to $A$ is the set $S$ defined so that $x S y$ if and only if there exists an element $x$ in $A$ such that $x R y$. 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...
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 \( x S y \) if \( x R y \) and \( x \neq y \), 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 \( x R y \) if \( x S y \) or \( x = y \), then \( R \) is the weak relation corresponding to \( S \). A relation \( R \) in \( X \) is reflexive if \( x R x \) for every element \( x \) of \( X \); it is symmetric if \( x R y \) implies that \( y R x \); and it is transitive if \( x R y \) and \( y R z \) imply that \( x R z \).

A function \( F \) is a relation, a subset of \( X \times Y \), 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 \( x F y \) and \( x F z \) 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 \( (x, y) \) in \( F \) or \( x F y \), 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 \( F_1 \) and \( F_2 \) is called the composite of \( F_1 \) and \( F_2 \) if the range of \( F_1 \) is a subset of the domain of \( F_2 \).

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 \( x R y \) 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 \((n\text{-ary})\) relation, and say that the relation is a subset of the Cartesian product \( X[1] \times ... \times X[n] \) where \( x[i] \) is an element of \( X[i] \), \( 1 <= i <= n \). The projection of an \( n\text{-ary} \) relation \( R \) onto coordinate \( i \) is the set \( \{x[i] : (x[1], ..., x[i], ..., x[n]) \in R \text{ for some } x[j] \in X[j], 1 <= j <= n \text{ and not } i = j \} \). 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 \( n\text{-ary} \) 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 \( Y[1] \times ... \times Y[n] \) to \( Z \). The relative product of \( TR \) and \( S \) is the binary relation \( T \) from \( X \) to \( Z \) defined so that \( x T z \) if and only if there exists an element \( y[i] \) in \( Y[i] \) for each \( 1 <= i <= n \) such that \( x R[i] y[i] \) and \( (y[1], ..., y[n]) \in S \). Now let \( TR \) be an ordered set \( (R[1], ..., R[n]) \) of binary relations from \( X[i] \) to \( Y[i] \) and \( S \) a subset of \( X[1] \times ... \times X[n] \). The multiple relative product of \( TR \) and \( S \) is defined to be the set \( \{z : z = ((x[1], ..., x[n]), (y[1], ..., y[n])) \text{ for some } (x[1], ..., x[n]) \in S \text{ and for some } (x[i], y[i]) \in R[i], 1 <= i <= n \} \). The natural join of an \( n\text{-ary} \) relation \( R \) and an \( m\text{-ary} \) relation \( S \) on
coordinate \( i \) and \( j \) is defined to be the set \( \{ z : z = (x[1], \ldots, x[n], y[1], \ldots, y[j-1], y[j+1], \ldots, y[m]) \} \) for some \( (x[1], \ldots, x[n]) \) in \( R \) and for some \( (y[1], \ldots, y[m]) \) in \( S \) such that \( x[i] = y[j] \).

The sets recognized by this module will be represented by elements of the relation \( \text{Sets} \), defined as the smallest set such that:

- for every atom \( T \) except \( '.' \) and for every term \( X \), \( (T, X) \) belongs to \( \text{Sets} \) (atomic sets);
- \( \{ [','], [] \} \) belongs to \( \text{Sets} \) (the untyped empty set);
- for every tuple \( T = \{ T[1], \ldots, T[n] \} \) and for every tuple \( X = \{ X[1], \ldots, X[n] \} \), if \( (T[i], X[i]) \) belongs to \( \text{Sets} \) for every \( 1 \leq i \leq n \) then \( (T, X) \) belongs to \( \text{Sets} \) (ordered sets);
- for every term \( T \), if \( X \) is the empty list or a non-empty sorted list \( [X[1], \ldots, X[n]] \) without duplicates such that \( (T, X[i]) \) belongs to \( \text{Sets} \) for every \( 1 \leq i \leq n \), then \( ([T], X) \) belongs to \( \text{Sets} \) (typed unordered sets).

An external set is an element of the range of \( \text{Sets} \). A type is an element of the domain of \( \text{Sets} \). If \( S \) is an element \( (T, X) \) of \( \text{Sets} \), 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 270] creates a set from a type and an Erlang term turned into an external set.

The actual sets represented by \( \text{Sets} \) are the elements of the range of the function \( \text{Set} \) from \( \text{Sets} \) to Erlang terms and sets of Erlang terms:

- \( \text{Set}(T, \text{Term}) = \text{Term} \), where \( T \) is an atom;
- \( \text{Set}(\{ T[1], \ldots, T[n] \}, \{ X[1], \ldots, X[n] \}) = (\text{Set}(T[1], X[1]), \ldots, \text{Set}(T[n], X[n])) \);
- \( \text{Set}([T], [X[1], \ldots, X[n]]) = (\text{Set}(T, X[1]), \ldots, \text{Set}(T, X[n])) \);
- \( \text{Set}([T], []) = [] \).

When there is no risk of confusion, elements of \( \text{Sets} \) 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 \( \text{Set}(U) \) is the union of \( \text{Set}(S1) \) and \( \text{Set}(S2) \).

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 277], checks that the arguments represent binary relations by matching \([\{A,B\}]\) against the type of the first argument \( \text{Arg1} \) say, and \([\{C,D\}]\) against the type of the second argument \( \text{Arg2} \) say. The fact that \([\{A,B\}]\) matches the type of \( \text{Arg1} \) is to be interpreted as \( \text{Arg1} \) representing a binary relation from \( X \) to \( Y \), where \( X \) is defined as all sets \( \text{Set}(x) \) for some element \( x \) in \( S \) the type of which is \( A \), and similarly for \( Y \). In the same way \( \text{Arg2} \) 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 \( \text{SetFun} \) in the following, can be specified as a functional object (fun), a tuple \( \{ \text{external, Fun} \} \), or an integer. If \( \text{SetFun} \) 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) \rightarrow \text{element}(I, X)\}, but is to be preferred since it makes it possible to handle this case even more efficiently. Examples of SetFuns:

\[
\{\text{sofs, union}\}
\]
\[
\text{fun}(S) \rightarrow \text{sofs:partition}(1, S) \text{ end}
\]
\[
\{\text{external, fun(A) \rightarrow A end}\}
\]
\[
\{\text{external, fun(\{A,\_,C\}) \rightarrow \{C,A\} end}\}
\]
\[
\{\text{external, fun(\_,\{\_,\_,\_\}) \rightarrow C end}\}
\]
\[
\{\text{external, fun(\_,\{\_,\_,\_\}=C) \rightarrow \{E,\{E,C\}\} end}\}
\]

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_software_set, to_external, type.

The functions of this module exit the process with a badarg, bad Functor, 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() \geq 1
  \mid \{\text{external, fun(external_set()) \rightarrow external_set()\}
  \mid \text{fun(anyset()) \rightarrow anyset()}
spec_fun() = \{\text{external, fun(external_set()) \rightarrow bool()}
  \mid \text{fun(anyset()) \rightarrow bool()}
type() = - a type -
Exports

\texttt{a_function(Tuples [, Type]) \rightarrow Function}

Types:
\begin{itemize}
  \item Function = \texttt{function()}
  \item Tuples = \texttt{[tuple()]} \hspace{1cm} \\
  \item Type = \texttt{type()}
\end{itemize}

Creates a function \cite{page 260}. \texttt{a_function(F, T)} is equivalent to \texttt{from_term(F, T)}, if the result is a function. If no type \cite{page 261} is explicitly given, \texttt{[[atom, atom]]} is used as type of the function.

\texttt{canonical_relation(SetOfSets) \rightarrow BinRel}

Types:
\begin{itemize}
  \item BinRel = \texttt{binary\_relation()}
  \item SetOfSets = \texttt{set\_of\_sets()}
\end{itemize}

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 \cite{page 260} of a set X and R is the equivalence relation in X induced by SetOfSets, then the returned relation is the canonical map \cite{page 260} from X onto the equivalence classes with respect to R.

\begin{verbatim}
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]]]
\end{verbatim}

\texttt{composite(Function1, Function2) \rightarrow Function3}

Types:
\begin{itemize}
  \item Function1 = Function2 = Function3 = \texttt{function()}
\end{itemize}

Returns the composite \cite{page 260} of the functions Function1 and Function2.

\begin{verbatim}
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]]
\end{verbatim}

\texttt{constant_function(Set, AnySet) \rightarrow Function}

Types:
\begin{itemize}
  \item AnySet = \texttt{anyset()}
  \item Function = \texttt{function()}
  \item Set = \texttt{set()}
\end{itemize}

Creates the function \cite{page 260} 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 259] 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 259] 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 260] 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 261] 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 259] 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 259] of BinRel1 to Set.

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.

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).

[1, 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 261]. 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 259] of BinRel1 such that for each element E in Set that does not belong to the domain [page 259] of BinRel1, BinRel2 contains the pair (E, AnySet).

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 260]. family(F, T) is equivalent to
from_term(F, T), if the result is a family. If no type [page 261] 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 260], 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 260] 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 259] 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 260] 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 260] 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 260] 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 259] 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 260], 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,[]],[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 260] 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 260] 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 259] of Family1[i].
family_specification(Fun, Family1) -> Family2

Types:
- Fun = spec_fun()
- Family1 = Family2 = family()

If Family1 is a family [page 260], then Family2 is the restriction [page 259] 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 261] 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 260] 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.

If 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 260], 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 260] 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 259] 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 260], 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 260] 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 261] ExternalSet and the type [page 261] Type. It is assumed that Type is a valid type [page 261] of ExternalSet.

```
from_sets(ListOfSets) -> Set
```
Types:
- Set = set()
- ListOfSets = [anyset()]

Returns the unordered set [page 261] 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 261] 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 261] by traversing the term Term, sorting lists,
removing duplicates and deriving or verifying a valid type [page 261] for the so
obtained external set. A explicitly given type [page 261] Type can be used to limit the
depth of the traversal; an atomic type stops the traversal, as demonstrated by this
element 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`.

```erlang
image(BinRel, Set1) -> Set2
Types:
  • BinRel = binary_relation()
  • Set1 = Set2 = set()
Returns the image [page 259] 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]
```

```erlang
intersection(SetOfSets) -> Set
Types:
  • Set = set()
  • SetOfSets = set_of_sets()
Returns the intersection [page 259] of the set of sets SetOfSets.
Intersecting an empty set of sets exits the process with a badarg message.
```

```erlang
intersection(Set1, Set2) -> Set3
Types:
  • Set1 = Set2 = Set3 = set()
Returns the intersection [page 259] of Set1 and Set2.
```

```erlang
intersection_of_family(Family) -> Set
Types:
  • Family = family()
  • Set = set()
Returns the intersection of the family [page 260] Family.
Intersecting an empty family exits the process with a badarg message.
```

```erlang
inverse(Function1) -> Function2
Types:
  • Function1 =Function2 = function()
Returns the inverse [page 260] of the function Function1.
```
inverse_image(BinRel, Set1) -> Set2
Types:
  BinRel = binary_relation()
  Set1 = Set2 = set()
Returns the inverse image [page 259] of Set1 under the binary relation BinRel.

is_a_function(BinRel) -> Bool
Types:
  Bool = bool()
  BinRel = binary_relation()
Returns true if the binary relation BinRel is a function [page 260] 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 259], 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 259], false otherwise.

is_set(AnySet) -> Bool
Types:
AnySet = anyset()
Bool = bool()

Returns true if AnySet is an unordered set [page 261], 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 261], 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 259] of Set2, false otherwise.

is_type(Term) -> Bool

Types:
  • Bool = bool()
  • Term = term()

Returns true if the term Term is a type [page 261].

join(Relation1, I, Relation2, J) -> Relation3

Types:
  • Relation1 = Relation2 = Relation3 = relation()
  • I = J = integer() > 0

Returns the natural join [page 260] of the relations Relation1 and Relation2 on coordinates I and J.

J = sofs:join(R1, 3, R2, 1),
sofs:to_external(J).

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 260] 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:
  - A Set = set() | ordset()
  - NoElements = integer() >= 0

Returns the number of elements of the ordered or unordered set A Set.

partition(SetOfSets) -> Partition

Types:
  - SetOfSets = set of sets()
  - Partition = set()

Returns the partition [page 260] 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 260] 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 260] 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.
STDLIB Reference Manual

STDLIB Reference Manual

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 260] Family where the indexed set is a partition [page 260] 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 260].

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,b}],[{a},{a,a,b,b},{a,b,b,b}]]

product(TupleOfSets) -> Relation

Types:
  • Relation = relation()
  • TupleOfSets = tuple-of(set())

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},{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 259] 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 260] 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 259] 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 259]. relation(R, T) is equivalent to from\_term(R, T), if T is a type [page 261] and the result is a relation. If Type is an integer N, then \([\text{atom}, \ldots, \text{atom}]\), 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 260] Family such that the index set is equal to the domain [page 259] of the binary relation BinRel, and Family[i] is the image [page 259] of the set of i under BinRel.
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 260] of the ordered set (R[1], ..., R[n]) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian product [page 260] of the ranges of R[i], range R[1] \times ... \times range R[n], is used instead (intuitively, nothing is “lost”).

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_product1(BinRel1, BinRel2) -> BinRel3

Types:
- BinRel1 = BinRel2 = BinRel3 = binary relation()

Returns the relative product [page 259] of the binary relations BinRel1 and BinRel2.

relative_product1(BinRel1, BinRel2) -> BinRel3

Types:
- BinRel1 = BinRel2 = BinRel3 = binary relation()

Returns the relative product [page 259] of the converse [page 259] of the binary relation BinRel1 and the binary relation BinRel2.

restriction(BinRel1, Set) -> BinRel2

Types:
- BinRel1 = BinRel2 = binary relation()
- Set = set()
Returns the restriction [page 259] 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(S, 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 261]. set(L, T) is equivalent to from_term(L, T), if the result is an unordered set. If no type [page 261] 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 261] 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 260] corresponding to the binary relation BinRel1.

```erlang
1> R1 = sofs:relation([[1,1],[1,2],[2,1],[2,2]]),
   R2 = sofs:strict_relation(R1),
   sofs:to_external(R2).

1,2
```

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.

```erlang
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 = fun(A) -> [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\}:

```erlang
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 259] of Set under BinRel is returned by this function:

```erlang
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:

```erlang
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()
sofs STDLIB Reference Manual

Returns the symmetric difference [page 259] (or the Boolean sum) of Set1 and Set2.

\[ S1 = \text{sofs:set([1,2,3])}, \]
\[ S2 = \text{sofs:set([2,3,4])}, \]
\[ P = \text{sofs:symdiff(S1, S2)} , \]
\[ \text{sofs:to_external}(P) . \]

\[ [1,4] \]

**symmetric_partition(Set1, Set2) -> \{Set3, Set4, Set5\}**

Types:
- \( \text{Set1} = \text{Set2} = \text{Set3} = \text{Set4} = \text{Set5} = \text{set}() \)

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:
- \( \text{ExternalSet} = \text{external_set() } \)
- \( \text{AnySet} = \text{anyset() } \)

Returns the external set [page 261] of an atomic, ordered or unordered set.

**to_sets(ASet) -> Sets**

Types:
- \( \text{A Set} = \text{set()} \mid \text{ordset()} \)
- \( \text{Sets} = \text{tuple_of(AnySet)} \mid [\text{AnySet}] \)

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:
- \( \text{AnySet} = \text{anyset()} \)
- \( \text{Type} = \text{type()} \)

Returns the type [page 261] of an atomic, ordered or unordered set.

**union(SetOfSets) -> Set**

Types:
- \( \text{Set} = \text{set()} \)
- \( \text{SetOfSets} = \text{set_of_sets()} \)

Returns the union [page 259] of the set of sets SetOfSets.

**union(Set1, Set2) -> Set3**

Types:
- \( \text{Set1} = \text{Set2} = \text{Set3} = \text{set}() \)
Returns the union [page 259] of Set1 and Set2.

\[
\text{union_of_family}(\text{Family}) \rightarrow \text{Set}
\]

Types:
- \text{Family} = \text{family()}
- \text{Set} = \text{set()}

Returns the union of the family [page 260] Family.

```.erlang
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]
```

\[
\text{weak_relation}(\text{BinRel1}) \rightarrow \text{BinRel2}
\]

Types:
- \text{BinRel1} = \text{BinRel2} = \text{binary_relation()}

Returns a subset S of the weak relation [page 260] W corresponding to the binary relation BinRel1. Let F be the field [page 260] of BinRel1. The subset S is defined so that \( x \in S \) if \( x \in W \) for some \( x \) in \( F \) and for some \( y \) in \( F \).

```erlang
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 76], digraph(3) [page 80], orddict(3) [page 219], ordsets(3) [page 220], sets(3) [page 244]
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

\[\text{span} \text{ :: } \text{String, Chars} \rightarrow \text{Length} \]
\[\text{cspan} \text{ :: } \text{String, Chars} \rightarrow \text{Length} \]

**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

\[\text{substr} \text{ :: } \text{String, Start} \rightarrow \text{SubString} \]
\[\text{substr} \text{ :: } \text{String, Start, Length} \rightarrow \text{Substring} \]

**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"

\[\text{tokens} \text{ :: } \text{String, SeparatorList} \rightarrow \text{Tokens} \]

**Types:**
- `String = SeparatorList = string()`
- `Tokens = [string()]`

Returns a list of tokens in `String`, separated by the characters in `SeparatorList`.

For example:

> `tokens("abc defxghix jkl", "x ").`

["abc", "def", "ghi", "jkl"]

\[\text{chars} \text{ :: } \text{Character, Number} \rightarrow \text{String} \]
\[\text{chars} \text{ :: } \text{Character, Number, Tail} \rightarrow \text{String} \]

**Types:**
- `Character = char()`
string

- 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 239]). 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.
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 \( \text{MaxR} \) and \( \text{MaxT} \). If more than \( \text{MaxR} \) restarts occur within \( \text{MaxT} \) seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

\[
\text{child_spec() = \{Id,StartFunc,Restart,Shutdown,Type,Modules\}}
\]

\[
\begin{align*}
\text{Id} &= \text{term()} \\
\text{StartFunc} &= \{\text{M,F,A}\} \\
\text{M} &= \text{F} = \text{atom()} \\
\text{A} &= \text{[term()]} \\
\text{Restart} &= \text{permanent | transient | temporary} \\
\text{Shutdown} &= \text{brutal_kill | int()>=0 | infinity} \\
\text{Type} &= \text{worker | supervisor} \\
\text{Modules} &= \text{[Module] | dynamic} \\
\text{Module} &= \text{atom()}
\end{align*}
\]

- \( \text{Id} \) is a name that is used to identify the child specification internally by the supervisor.
- \( \text{StartFunc} \) defines the function call used to start the child process. It should be a module-function-arguments tuple \( \{\text{M,F,A}\} \) used as \( \text{apply(M,F,A)} \).

The start function must create and link to the child process, and should return \{\text{ok,Child}\} or \{\text{ok,Child,Info}\} where \text{Child} is the pid of the child process and \text{Info} an arbitrary term which is ignored by the supervisor.

If something goes wrong, the function may also return an error tuple \{\text{error,Error}\}.

Note that the start link functions of the different behaviour modules fulfill the above requirements.

- \( \text{Restart} \) 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.

- \( \text{Shutdown} \) defines how a child process should be terminated. \text{brutal_kill} means the child process will be unconditionally terminated using \text{exit(Child,kill)}. An integer timeout value means that the supervisor will tell the child process to terminate by calling \text{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 \text{exit(Child,kill)}.

If the child process is another supervisor, \( \text{Shutdown} \) should be set to \text{infinity} to give the subtree ample time to shutdown.

- \( \text{Type} \) 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:
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.
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`.

```erlang
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`

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`.

```erlang
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()`

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`.

See `start_child/2` for a description of `SupRef`. 

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`.

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) \rightarrow \{[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]) \rightarrow 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 152], gen_fsm(3) [page 161], gen_server(3) [page 171], sys(3) [page 297]
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 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**

```erlang
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`.

```erlang
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 287], sys(3) [page 297]
This module contains functions for sending system messages used by programs, and messages 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, P, 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 = {Year, Month, Day}
Time1 = {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.

cchange_code(Name, Module, OldVsn, Extra)
cchange_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.

gget_status(Name)
gget_status(Name, Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}

Types:
- PDict = [{Key, Value}]
- SysState = running | suspended
- Parent = pid()
* D bg = {dbg_opt()}
* Misc = term()

Gets the status of the process.

```lisp
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.

```lisp
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 \(\text{Msg}\) and \(\text{From}\) to this function.

This function never returns. It calls the function \texttt{Module:system\_continue(Parent, NDebug, Misc)} where the process continues the execution, or \texttt{Module:system\_terminate(Reason, Parent, Debug, Misc)} if the process should terminate. The Module must export \texttt{system\_continue/3}, \texttt{system\_terminate/4}, and \texttt{system\_code\_change/4} (see below).

The \texttt{Misc} argument can be used to save internal data in a process, for example its state. It is sent to \texttt{Module:system\_continue/3} or \texttt{Module:system\_terminate/4}.

\texttt{print\_log(Debug) -> void()}

Types:
- \texttt{Debug = \{dbg opt\}}

Prints the logged system events in the debug structure using \texttt{FormFunc} as defined when the event was generated by a call to \texttt{handle\_debug/4}.

\texttt{Mod:system\_continue(Parent, Debug, Misc)}

Types:
- \texttt{Parent = pid()}
- \texttt{Debug = \{dbg opt\}}
- \texttt{Misc = term()}

This function is called from \texttt{sys:handle\_system\_msg/6} when the process should continue its execution (for example after it has been suspended). This function never returns.

\texttt{Mod:system\_terminate(Reason, Parent, Debug, Misc)}

Types:
- \texttt{Reason = term()}
- \texttt{Parent = pid()}
- \texttt{Debug = \{dbg opt\}}
- \texttt{Misc = term()}

This function is called from \texttt{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.

\texttt{Mod:system\_code\_change(Misc, Module, OldVsn, Extra) -> \{ok, NMisc\}}

Types:
- \texttt{Misc = term()}
- \texttt{OldVsn = undefined | term()}
- \texttt{Module = atom()}
- \texttt{Extra = term()}
- \texttt{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.
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

\[
\text{start()} \rightarrow \text{ok}
\]

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.

\[
\text{apply\_after(Time, Module, Function, Arguments) \rightarrow \{ok, TRef\} | \{error, Reason\}}
\]

Types:
- \(\text{Time} = \text{integer()}\) in Milliseconds
- \(\text{Module} = \text{Function} = \text{atom()}\)
- \(\text{Arguments} = \text{[term()]}\)

Evaluates \text{apply(M, F, A) after Time amount of time has elapsed}. Returns \{ok, TRef\}, or \{error, Reason\}.

\[
\text{send\_after(Time, Pid, Message) \rightarrow \{ok, TRef\} | \{error, Reason\}}
\]

\[
\text{send\_after(Time, Message) \rightarrow \{ok, TRef\} | \{error, Reason\}}
\]

Types:
- \(\text{Time} = \text{integer()}\) in Milliseconds
- \(\text{Pid} = \text{pid()} \mid \text{atom()}\)
- \(\text{Message} = \text{term()}\)
- \(\text{Result} = \{\text{ok, TRef}\} \mid \{\text{error, Reason}\}\)

\text{send\_after/3} Evaluates \text{Pid ! Message after Time amount of time has elapsed}. \text{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
Suspended 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 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:

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Registry key</th>
</tr>
</thead>
<tbody>
<tr>
<td>hkcr</td>
<td>HKEY_CLASSES_ROOT</td>
</tr>
<tr>
<td>cu</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>hklm</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>users</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>hku</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>current config</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>hkc</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>dyn</td>
<td>HKEY_DYN_DATA</td>
</tr>
<tr>
<td>hkd</td>
<td>HKEY_DYN_DATA</td>
</tr>
</tbody>
</table>

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

\[\text{change_key(RegHandle, Key)} \rightarrow \text{ReturnValuen} \]

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 \\.

\[\text{change_key_create(RegHandle, Key)} \rightarrow \text{ReturnValuen} \]

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.

\[\text{close(RegHandle)} \rightarrow \text{ReturnValuen} \]

Types:
- RegHandle = term()

Closes the registry. After that, the RegHandle cannot be used.

\[\text{current_key(RegHandle)} \rightarrow \text{ReturnValuen} \]

Types:
- RegHandle = term()
- ReturnValuen = \{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).

\[\text{delete_key(RegHandle)} \rightarrow \text{ReturnValuen} \]

Types:
- RegHandle = term()
- ReturnValuen = 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 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(ErrId) -> ErrorString`

Types:
- `ErrId = atom()`
- `ErrorString = string()`

Convert an POSIX errorcode to a string (by calling `erl_posix_msg:msg`).

`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.

\[
\begin{align*}
a\_function/2 & \quad \text{sofs, 263} \\
abcast/2 & \quad \text{gen\_server, 174} \\
abcast/3 & \quad \text{gen\_server, 174} \\
absname/1 & \quad \text{filename, 140} \\
absname/2 & \quad \text{filename, 140} \\
abstract/1 & \quad \text{erl\_parse, 103} \\
acos/1 & \quad \text{math, 206} \\
\text{acos\_1} & \quad \text{math, 206} \\
\text{add/2} & \quad \text{gb\_sets, 146} \\
\text{add/3} & \quad \text{erl\_tar, 111} \\
\text{add/4} & \quad \text{erl\_tar, 111} \\
\text{add\_binding/3} & \quad \text{erl\_eval, 94} \\
\text{add\_edge/3} & \quad \text{digraph, 80} \\
\text{add\_edge/4} & \quad \text{digraph, 80} \\
\text{add\_edge/5} & \quad \text{digraph, 80} \\
\text{add\_element/2} & \quad \text{sets, 245} \\
\text{add\_handler/2} & \quad \text{gen\_event, 153} \\
\text{add\_sup\_handler/3} & \quad \text{gen\_event, 154} \\
\text{add\_vertex/1} & \quad \text{digraph, 81} \\
\text{add\_vertex/2} & \quad \text{digraph, 81} \\
\text{add\_vertex/3} & \quad \text{digraph, 81} \\
\text{all/0} & \quad \text{dets, 62} \\
\text{all/2} & \quad \text{ets, 117} \\
\text{any/2} & \quad \text{lists, 201} \\
\text{apply\_after/4} & \quad \text{timer, 304} \\
\text{apply\_interval/4} & \quad \text{timer, 305} \\
\text{arith\_op/2} & \quad \text{erl\_internal, 97} \\
\text{asin/1} & \quad \text{STDLIB, 313}
\end{align*}
\]
math, 206
  asinh/1
math, 206
atan/1
  math, 206
atan2/2
  math, 206
atanh/1
  math, 206
attach/1
  pool, 222
attribute/1
  erl_pp, 105
attribute/2
  erl_pp, 105
balance/1
  gb_sets, 146
gb_trees, 150
basename/1
  filename, 141
basename/2
  filename, 141
bchunk/2
dets, 62
beam_lib
  chunks/2, 48
cmp/2, 48
cmp_dirs/2, 49
diff_dirs/2, 49
format_error/1, 50
info/1, 48
strip/1, 49
strip_files/1, 49
strip_release/1, 50
version/1, 48
bif/2
erl_internal, 97
binding/2
erl_eval, 94
bindings/1
erl_eval, 94
bool_op/2
erl_internal, 97
bt/1
  c, 51
  bt/1, 51
c/1, 51
c/2, 51
cd/1, 51
flush/0, 52
help/0, 52
i/0, 52
i/3, 52
l/1, 52
c/1, 52
ls/0, 52
ls/1, 52
m/0, 52
m/1, 53
memory/0, 53
memory/1, 53
nc/1, 53
nc/2, 53
ni/0, 53
nl/1, 53
nregs/0, 53
pid/3, 53
pwd/0, 54
q/0, 54
regs/0, 54
xm/1, 54
zi/0, 54
c/1
c, 51
c/2
c, 51
calendar
date_to_gregorian_days/1, 55
date_to_gregorian_days/3, 55
datetime_to_gregorian_seconds/1, 55
day_of_the_week/1, 56
day_of_the_week/3, 56
gregorian_days_to_date/1, 56
gregorian_seconds_to_datetime/1, 56
is_leap_year/1, 56
last_day_of_the_month/2, 56
local_time/0, 57
local_time_to_universal_time/2, 57
local_time_to_universal_time_dst/1, 57
now_to_datetime/1, 58
now_to_local_time/1, 57
now_to_universal_time/1, 58
math, 206
create/1
  pg, 221
create/2
erl_tar, 111
  pg, 221
create/3
erl_tar, 112
cspan/2
  string, 283
current_key/1
  win32reg, 309
cyclic_strong_components/1
digraph_utils, 88
daeh/1
  queue, 233
date_to_gregorian_days/1
calendar, 55
date_to_gregorian_days/3
calendar, 55
datetime_to_gregorian_seconds/1
calendar, 55
day_of_the_week/1
calendar, 56
day_of_the_week/3
calendar, 56
debug_options/1
  sys, 301
deep_char_list/1
  io_lib, 191
del_binding/2
erl_eval, 94
del_edge/2
digraph, 81
del_edges/2
digraph, 81
del_element/2
  sets, 245
del_path/3
digraph, 81
del_vertex/2
digraph, 82
del_vertices/2
digraph, 82
delete/1
digraph, 82
  ets, 117
delete/2
dets, 63
  ets, 117
  gb_sets, 146
  gb_trees, 150
  lists, 194
  proplists, 228
delete_all_objects/1
dets, 63
  ets, 117
delete_any/2
  gb_sets, 146
  gb_trees, 150
delete_child/2
  supervisor, 291
delete_handler/3
gen_event, 155
delete_key/1
  win32reg, 309
delete_object/2
dets, 63
  ets, 117
delete_value/2
  win32reg, 310
dets
  all/0, 62
  bchunk/2, 62
  close/1, 63
  delete/2, 63
  delete_all_objects/1, 63
  delete_object/2, 63
  first/1, 63
  foldl/3, 64
  foldr/3, 64
  from_ets/2, 64
  info/1, 64
  info/2, 65
  init_table/3, 65
  insert/2, 66
  is_compatible_bchunk_format/2, 66
  is_dets_file/1, 67
  lookup/2, 67
  match/1, 67

316  STDLIB
match/2, 67
match/3, 68
match_delete/2, 68
match_object/1, 68
match_object/2, 69
match_object/3, 69
member/2, 69
next/2, 70
open_file/1, 70
open_file/2, 70
pid2name/1, 71
safe_fixtable/2, 71
select/1, 72
select/2, 72
select/3, 72
select_delete/2, 73
slot/2, 73
sync/1, 73
to_ets/2, 73
traverse/2, 74
update_counter/3, 74
dict
append/3, 76
append_list/3, 76
erase/2, 76
fetch/2, 76
fetch_keys/1, 77
filter/2, 77
find/2, 77
fold/3, 77
from_list/1, 77
is_key/2, 77
map/2, 77
merge/3, 78
new/0, 78
store/3, 78
to_list/1, 78
update/3, 78
update/4, 79
update_counter/3, 79
diff_dirs/2
beam_lib, 49
difference/2
gb_sets, 146
sofs, 264
digraph
add_edge/3, 80
add_edge/4, 80
add_edge/5, 80
add_vertex/1, 81
add_vertex/2, 81
add_vertex/3, 81
del_edge/2, 81
del_edges/2, 81
del_path/3, 81
del_vertex/2, 82
del_vertices/2, 82
del/1, 82
del/2, 82
del/edges/1, 82
del/edges/2, 82
get_cycle/2, 83
get_cycle/3, 83
get_short_cycle/2, 83
get_short_cycle/3, 83
in_degree/2, 84
in_edges/2, 84
in_neighbours/2, 84
info/1, 84
new/0, 85
new/1, 85
no_edges/1, 85
no_neighbours/2, 85
out_degree/2, 85
out_edges/2, 85
out_neighbours/2, 85
vertex/2, 86
vertices/1, 86
digraph_to_family/2
sofs, 264
digraph_utils
components/1, 88
condensation/1, 88
cyclic_strong_components/1, 88
is_acyclic/1, 88
loop_vertices/1, 88
postorder/1, 89
preorder/1, 89
reachable/2, 89
reachable_neighbours/2, 89
reaching/2, 89
reaching_neighbours/2, 89
strong_components/1, 90
subgraph/3, 90
toporder/1, 90
dirname/1
filename, 141
domain/1
sofs, 264
drestriction/2
sofs, 264

STDLIB

317
function/2, 105
guard/1, 105
guard/2, 105
erl_scan
  format_error/1, 109
  reserved_word/1, 109
  string/1, 108
  string/2, 108
  tokens/3, 108
erl_tar
  add/3, 111
  add/4, 111
  close/1, 111
  create/2, 111
  create/3, 112
  extract/1, 112
  extract/2, 112
  format_error/1, 113
  open/2, 113
  t/1, 114
  table/1, 114
  table/2, 114
  tt/1, 114
error_message/2
  lib , 192
esend/2
  pg , 221
ets
  all/0, 117
  delete/1, 117
  delete/2, 117
  delete_all_objects/1, 117
  delete_object/2, 117
  file2tab/1, 117
  first/1, 117
  fixtable/2, 118
  foldl/3, 118
  foldr/3, 118
  from_dets/2, 118
  fun2ms/1, 119
  i/0, 120
  i/1, 120
  info/1, 120
  info/2, 121
  init_table/2, 121
  insert/2, 121
  last/1, 122
  lookup/2, 122
  lookup_element/3, 122
  match/1, 123
match/2, 122
match/3, 123
match_delete/2, 124
match_object/1, 124
match_object/2, 124
match_object/3, 124
member/2, 124
new/2, 125
next/2, 125
prev/2, 126
rename/2, 126
safe_fixtable/2, 126
select/1, 129
select/2, 127
select/3, 128
select_count/2, 129
select_delete/2, 129
slot/2, 130
tab2file/2, 130
tab2list/1, 130
test_ms/2, 130
to_dets/2, 131
update_counter/3, 131
update_counter/4, 131
update_counter/6, 131
exit_after/2
  timer , 305
exit_after/3
  timer , 305
exp/1
  math , 206
expand/1
  win32reg , 310
expand/2
  proplists , 228
expr/1
  erl_pp , 106
expr/2
  erl_eval , 93
  erl_pp , 106
expr/3
  erl_eval , 93
  erl_pp , 106
expr/4
  erl_eval , 93
  erl_pp , 106
expr_list/2
  erl_eval , 94
expr_list/3
  erl_eval, 94
expr_list/4
  erl_eval, 94
exprs/1
  erl_pp, 106
exprs/2
  erl_eval, 93
  erl_pp, 106
exprs/3
  erl_eval, 93
  erl_pp, 106
exprs/4
  erl_eval, 93
extension/1
  filename, 141
extension/3
  sofs, 265
extract/1
  erl_tar, 112
extract/2
  erl_tar, 112
family/2
  sofs, 266
family_difference/2
  sofs, 266
family_domain/1
  sofs, 266
family_field/1
  sofs, 266
family_intersection/1
  sofs, 267
family_intersection/2
  sofs, 267
family_projection/2
  sofs, 267
family_range/1
  sofs, 267
family_specification/2
  sofs, 268
family_to_digraph/2
  sofs, 268
family_to_relation/1
  sofs, 268
family_union/1
  sofs, 269
family_union/2
  sofs, 269
fetch/2
  dict, 76
fetch_keys/1
  dict, 77
field/1
  sofs, 269
file2tab/1
  ets, 117
file_size/1
  filelib, 137
file_sorter
  check/1, 136
  check/2, 136
  keycheck/2, 136
  keycheck/3, 136
  keymerge/3, 136
  keymerge/4, 136
  keysort/2, 135
  keysort/3, 135
  keysort/4, 135
  merge/2, 135
  merge/3, 135
  sort/1, 135
  sort/2, 135
  sort/3, 135
filelib
  ensure_dir/1, 137
  file_size/1, 137
  fold_files/5, 137
  is_dir/1, 137
  is_file/1, 137
  last_modified/1, 138
  wildcard/1, 138
  wildcard/2, 138
filename
  absname/1, 140
  absname/2, 140
  basename/1, 141
  basename/2, 141
  dirname/1, 141
  extension/1, 141
  find_src/1, 143
  find_src/2, 143
from_ordset/1
gb_sets, 147
from_sets/1
sofs, 269, 270
from_term/2
sofs, 270
fun2ms/1
ets, 119
function/1
erlpp, 105
function/2
erlpp, 105
fwrite/1
io, 181
fwrite/2
io_lib, 189
fwrite/3
io, 181
gb_sets
add/2, 146
balance/1, 146
delete/2, 146
delete_any/2, 146
difference/2, 146
empty/0, 145
filter/2, 147
fold/3, 148
from_list/1, 147
from_ordset/1, 147
insert/2, 146
intersection/1, 146
intersection/2, 146
is_empty/1, 145
is_member/2, 145
is_set/1, 148
is_subset/2, 147
iterator/1, 147
largest/1, 147
next/1, 147
singleton/1, 145
size/1, 145
smallest/1, 147
take_largest/1, 147
take_smallest/1, 147
to_list/1, 147
union/1, 146
union/2, 146
gb_trees
balance/1, 150
delete/2, 150
delete_any/2, 150
empty/0, 149
enter/3, 150
from_orddict/1, 150
get/2, 149
insert/3, 150
is_defined/2, 150
is_empty/1, 149
iterator/1, 151
keys/1, 150
largest/1, 151
lookup/2, 149
next/1, 151
size/1, 149
smallest/1, 151
take_largest/1, 151
take_smallest/1, 151
to_list/1, 150
update/3, 150
values/1, 150
gen_event
add_handler/3, 153
add_sup_handler/3, 154
call/3, 155
call/4, 155
delete_handler/3, 155
Module:code_change/3, 160
Module:handle_call/2, 159
Module:handle_event/2, 158
Module:handle_info/2, 159
Module:init/1, 158
Module:terminate/2, 159
notify/2, 154
start/0, 153
start/1, 153
start_link/0, 153
start_link/1, 153
stop/1, 157
swap_handler/5, 156
swap_sup_handler/5, 157
sync_notify/2, 154
which_handlers/1, 157
gen_fsm
cancel_timer/1, 165
Module:code_change/4, 169
Module:handle_event/3, 167
Module:handle_info/3, 168
Module:handle_sync_event/4, 168
Module:init/1, 166
Module:StateName/2, 166
Module:StateName/3, 167
Module:terminate/3, 169
reply/2, 164
send_all_state_event/2, 163
send_event/2, 163
send_event_after/2, 164
start/3, 162
start/4, 162
start_link/3, 162
start_link/4, 162
start_timer/2, 165
sync_send_all_state_event/2, 164
sync_send_all_state_event/3, 164
sync_send_event/2, 163
sync_send_event/3, 163
gen_server
abcast/2, 174
abcast/3, 174
call/2, 172
call/3, 172
cast/2, 174
enter_loop/4, 175
enter_loop/5, 175
Module:code_change/3, 178
Module:handle_call/3, 176
Module:handle_cast/2, 177
Module:handle_info/2, 177
Module:init/1, 176
Module:terminate/2, 178
multi_call/2, 173
multi_call/3, 173
multi_call/4, 173
reply/2, 175
start/3, 171
start/4, 171
start_link/3, 171
start_link/4, 171
get/2
 gb_trees, 149
get_all_values/2
 proplists, 229
get_bool/2
 proplists, 229
get_chars/3
 io, 180
get_cycle/2
digraph, 83
get_debug/3
printio, 3
stdlib
get_keys/1
 proplists, 229
get_line/2
 io, 180
get_node/0
 pool, 223
get_nodes/0
 pool, 223
get_path/3
digraph, 83
gregorian_days_to_date/1
calendar, 56
gregorian_seconds_to_datetimen/1
calendar, 56
gsub/3
 regexp, 240
guard/1
 erl_pp, 105
guard/2
 erl_pp, 105
guard_bif/2
 erl_internal, 97
handle_debug/1
 sys, 301
handle_system_msg/6
 sys, 301
head/1
 queue, 233
help/0
c, 52
history/1
  shell, 254

hms/3
timer, 306

hours/1
timer, 306

i/0
c, 52
  ets, 120

i/1
  ets, 120

i/3
c, 52
  ets, 120

image/2
  sofs, 271

in/2
  queue, 233

in_degree/2
  digraph, 84

in_edges/2
  digraph, 84

in_neighbours/2
  digraph, 84

in_r/2
  queue, 234

indentation/2
  io_lib, 190

info/1
  beam_lib, 48
    dets, 64
    digraph, 84
    ets, 120

info/2
  dets, 65
  ets, 121

init/1
  queue, 234

init/3
  log_mfh, 205

init/4
  log_mfh, 205

init_ack/1
  proc_lib, 226

init_ack/2
  proc_lib, 226

init_table/2
  ets, 121

init_table/3
  dets, 65

initial_call/1
  proc_lib, 227

insert/2
  dets, 66
  ets, 121
  gb_sets, 146

insert/3
  gb_trees, 150

install/3
  sys, 300

install/4
  sys, 300

intersection/1
  gb_sets, 146
  sets, 245
  sofs, 271

intersection/2
  gb_sets, 146
  sets, 245
  sofs, 271

intersection_of_family/1
  sofs, 271

inverse/1
  sofs, 271

inverse_image/2
  sofs, 272

io
  format/1, 181
  format/3, 181
  fread/3, 185
  fwrite/1, 181
  fwrite/3, 181
  get_chars/3, 180
  get_line/2, 180
  nl/1, 180
  parse_erl_exprs/1, 187
  parse_erl_exprs/3, 187
  parse_erl_form/1, 187
  parse_erl_form/3, 187

324

STDLib
seq/2, 198
seq/3, 198
sort/1, 198
sort/2, 198
split/2, 203
splitwith/2, 203
sublist/2, 198
sublist/3, 199
subtract/2, 199
suffix/2, 199
sum/1, 199
takewhile/2, 203
ukeymerge/3, 199
ukeysort/2, 199
umerge/1, 200
umerge/2, 200
umerge/3, 200
umerge3/3, 200
usort/1, 200
usort/2, 200

local_time/0
calendar, 57
local_time_to_universal_time/2
calendar, 57
local_time_to_universal_time_dst/1
calendar, 57

log/1
math, 206
log/2
sys, 298
log/3
sys, 298
log10/1
math, 206
loghf
init/3, 205
init/4, 205
log_to_file/2
sys, 298
log_to_file/3
sys, 298
lookup/2
dets, 67
ets, 122
gb_trees, 149
proplists, 230
lookup_all/2
proplists, 230
lookup_element/3
ets, 122
loop_vertices/1
digraph_utils, 88
ls/0
c, 52
ls/1
c, 52
m/0
c, 52
m/1
c, 53
map/2
dict, 77
lists, 202
mapfoldl/3
lists, 202
mapfoldr/3
lists, 203
match/1
dets, 67
ets, 123
match/2
dets, 67
ets, 122
regexp, 239
match/3
dets, 68
ets, 123
match_delete/2
dets, 68
ets, 124
match_object/1
dets, 68
ets, 124
match_object/2
dets, 69
ets, 124
match_object/3
dets, 69
ets, 124
matches/2
regexp, 239
math
  acos/1, 206
  acosh/1, 206
  asin/1, 206
  asinh/1, 206
  atan/1, 206
  atan2/2, 206
  atanh/1, 206
  cos/1, 206
  cosh/1, 206
  erf/1, 206
  erfc/1, 207
  exp/1, 206
  log/1, 206
  log10/1, 206
  pi/0, 206
  pow/2, 206
  sin/1, 206
  sinh/1, 206
  sqrt/1, 206
  tan/1, 206
  tanh/1, 206

max/1
  lists, 196

member/2
  dets, 69
  ets, 124
  lists, 196

members/1
  pg, 221

memory/0
  c, 53

memory/1
  c, 53

merge/1
  lists, 196

merge/2
  file_sorter, 135
  lists, 196

merge/3
  dict, 78
  file_sorter, 135
  lists, 196

merge/3/3
  lists, 197

min/1
  lists, 197

minutes/1

timer, 306

Mod:system_code_change/4
  sys, 302

Mod:system_continue/3
  sys, 302

Mod:system_terminate/4
  sys, 302

module/1
  erl_lint, 99

module/2
  erl_lint, 99

module/3
  erl_lint, 99

Module:code_change/3
  gen_event, 160
  gen_server, 178

Module:code_change/4
  gen_fsm, 169

Module:handle_call/2
  gen_event, 159

Module:handle_call/3
  gen_server, 176

Module:handle_cast/2
  gen_server, 177

Module:handle_event/2
  gen_event, 158

Module:handle_event/3
  gen_fsm, 167

Module:handle_info/2
  gen_event, 159
  gen_server, 177

Module:handle_info/3
  gen_fsm, 168

Module:handle_sync_event/4
  gen_fsm, 168

Module:init/1
  gen_event, 158
  gen_fsm, 166
  gen_server, 176
  supervisor, 293
  supervisor_bridge, 295

Module:StateName/2
  gen_fsm, 166

Module:StateName/3
gen_fsm, 167
Module: terminate/2
gen_event, 159
gen_server, 178
supervisor_bridge, 295
Module: terminate/3
gen_fsm, 169
ms_transform
format_error/1, 218
parse_transform/2, 217
transform_from_shell/3, 217
multi_call/2
gen_server, 173
multi_call/3
gen_server, 173
multi_call/4
gen_server, 173
multiple_relative_product/2
sofs, 273
	native_name/1
filename, 142
nc/1
c, 53
nc/2
c, 53
new/0
dict, 78
digraph, 85
queue, 235
sets, 244
new/1
digraph, 85
new/2
ets, 125
new_bindings/0
erl_eval, 94
new_node/2
pool, 223
next/1
gb_sets, 147
gb_trees, 151
next/2
dets, 70
ets, 125
ni/0
c, 53
nl/0
io_lib, 189
nl/1
c, 53
io, 180
no_debug/1
sys, 299
no_debug/2
sys, 299
no_edges/1
digraph, 85
no_elements/1
sofs, 274
no_vertices/1
digraph, 85
nonl/1
lib, 192
normalise/1
erl_parse, 103
normalize/2
proplists, 230
notify/2
gen_event, 154
now_diff/2
timer, 306
now_to_datetime/1
calendar, 58
now_to_local_time/1
calendar, 57
now_to_universal_time/1
calendar, 58
nregs/0
c, 53
nth/2
lists, 197
nthtail/2
lists, 197
op_type/2
erl_internal, 98
open/1
win32reg, 310

STDLIB 329
print_log/1
sys, 302
printable_list/1
io/lib, 191
proc_lib
   format/1, 226
   init_ack/1, 226
   init_ack/2, 226
   initial_call/1, 227
   spawn/1, 224
   spawn/2, 224
   spawn/3, 224
   spawn/4, 224
   spawn_link/1, 224
   spawn_link/2, 224
   spawn_link/3, 224
   spawn_link/4, 224
   spawn_opt/2, 225
   spawn_opt/3, 225
   spawn_opt/4, 225
   spawn_opt/5, 225
   start/3, 225
   start/4, 225
   start/5, 225
   start_link/3, 225
   start_link/4, 225
   start_link/5, 225
   translate_initial_call/1, 227
product/1
   sofs, 275
product/2
   sofs, 275
program/0
   lib, 192
projection/2
   sofs, 276
property/1
   proplists, 231
property/2
   proplists, 231
proplists
   append_values/2, 228
   compact/1, 228
   delete/2, 228
   expand/2, 228
   get_all_values/2, 229
   get_bool/2, 229
   get_keys/1, 229
   get_value/2, 230
   get_value/3, 230
   is_defined/2, 230
   lookup/2, 230
   lookup_all/2, 230
   normalize/2, 230
   property/1, 231
   property/2, 231
   split/2, 231
   substitute_aliases/2, 232
   substitute_negations/2, 232
   unfold/1, 232
pseudo/1
   slave, 258
pseudo/2
   slave, 258
pspawn/3
   pool, 223
pspawn_link/3
   pool, 223
put_chars/2
   io, 180
pwd/0
   c, 54
   q/0
   c, 54
queue
   cons/2, 233
   daeh/1, 233
   from_list/1, 233
   head/1, 233
   in/2, 233
   in_r/2, 234
   init/1, 234
   is_empty/1, 234
   join/2, 234
   lait/1, 234
   last/1, 234
   len/1, 234
   new/0, 235
   out/1, 235
   out_r/1, 235
   reverse/1, 235
   snoc/2, 235
   split/2, 235
   tail/1, 235
   to_list/1, 236
random

STDLIB 331
seed/0, 237
seed/3, 237
seed0/0, 237
uniform/0, 237
uniform/1, 237
uniform_s/1, 238
uniform_s/2, 238
range/1
  sofs, 276
rchr/2
  string, 282
reachable/2
digraph_utils, 89
reachable_neighbours/2
digraph_utils, 89
reaching/2
digraph_utils, 89
reaching_neighbours/2
digraph_utils, 89
read/2
  io, 181
regexp
  first_match/2, 239
  format_error/1, 241
gsub/3, 240
  match/2, 239
  matches/2, 239
  parse/1, 241
  sh_to_awk/1, 241
  split/2, 240
  sub/3, 240
regs/0
  c, 54
relation/2
  sofs, 276
relation_to_family/1
  sofs, 276
relative_product/2
  sofs, 277
relative_product1/2
  sofs, 277
relay/1
  slave, 258
remove/2
  sys, 300
remove/3
  sys, 300
rename/2
  ets, 126
reply/2
  gen_fsm, 164
  gen_server, 175
reserved_word/1
erl_scan, 109
restart_child/2
  supervisor, 291
restriction/2
  sofs, 277
restriction/3
  sofs, 278
results/1
  shell, 254
resume/1
  sys, 299
resume/2
  sys, 299
reverse/1
  lists, 197
  queue, 235
reverse/2
  lists, 198
right/2
  string, 285
right/3
  string, 285
rootname/1
  filename, 143
rootname/2
  filename, 143
rstr/2
  string, 282
safe_fixtable/2
dets, 71
  ets, 126
scan_erl_exprs/1
  io, 186
scan_erl_exprs/3
  io, 186
math, 206

size/1
gb_sets, 145
gb_trees, 149
sets, 244

slave
pseudo/1, 258
pseudo/2, 258
relay/1, 258
start/1, 256
start/2, 256
start/3, 257
start_link/1, 256
start_link/2, 257
start_link/3, 257
stop/1, 258

sleep/1
timer, 305

slot/2
dets, 73
ets, 130

smallest/1
gb_sets, 147
gb_trees, 151

snoc/2
queue, 235

sofs
a_function/2, 263
canonical_relation/1, 263
composite/2, 263
counter_function/2, 263
domain/1, 264
degree/2, 264
degree/3, 265
degree_set, 265
extension/3, 265
family/2, 266
family_difference/2, 266
family_domain/1, 266
family_field/1, 266
family_intersection/1, 267
family_intersection/2, 267
family_projection/2, 267
family_range/1, 267
family_specification/2, 268
family_to_digraph/2, 268
family_to_relation/1, 268
family_union/1, 269
family_union/2, 269
field/1, 269
from_external/2, 269
from_sets/1, 269, 270
from_term/2, 270
image/2, 271
intersection/1, 271
intersection/2, 271
intersection_of_family/1, 271
inverse/1, 271
inverse_image/2, 272
is_a_function/1, 272
is_disjoint/2, 272
is_empty_set/1, 272
is_equal/2, 272
is_set/1, 272
is_sofs_set/1, 273
is_subset/2, 273
is_type/1, 273
join/4, 273
multiple_relative_product/2, 273
no_elements/1, 274
partition/1, 274
partition/2, 274
partition/3, 274
partition_family/2, 275
product/1, 275
product/2, 275
projection/2, 276
range/1, 276
relation/2, 276
relation_to_family/1, 276
relative_product/2, 277
relative_product/1, 277
restriction/2, 277
restriction/3, 278
set/2, 278
specification/2, 278
strict_relation/1, 278
substitution/2, 279
symdiff/2, 279
symmetric_partition/2, 280
to_external/1, 280
to_sets/1, 280
type/1, 280
union/1, 280
union/2, 280
union_of_family/1, 281
weak_relation/1, 281

sort/1
file_sorter, 135
lists, 198

sort/2
  file_sorter, 135
  lists, 198

sort/3
  file_sorter, 135

span/2
  string, 283

spawn/1
  proc_lib, 224

spawn/2
  proc_lib, 224

spawn/3
  proc_lib, 224

spawn/4
  proc_lib, 224

spawn_link/1
  proc_lib, 224

spawn_link/2
  proc_lib, 224

spawn_link/3
  proc_lib, 224

spawn_link/4
  proc_lib, 224

spawn_opt/2
  proc_lib, 225

spawn_opt/3
  proc_lib, 225

spawn_opt/4
  proc_lib, 225

spawn_opt/5
  proc_lib, 225

specification/2
  sofs, 278

split/1
  filename, 143

split/2
  lists, 203
  proplists, 231
  queue, 235
  regexp, 240

splitwith/2
  lists, 203

sqrt/1
  math, 206

start/0
  gen_event, 153
  timer, 304

start/1
  gen_event, 153
  pool, 222
  slave, 256

start/2
  pool, 222
  slave, 256

start/3
  gen_fsm, 162
  gen_server, 171
  proc_lib, 225
  slave, 257

start/4
  gen_fsm, 162
  gen_server, 171
  proc_lib, 225

start/5
  proc_lib, 225

start_child/2
  supervisor, 289

start_link/0
  gen_event, 153

start_link/1
  gen_event, 153
  slave, 256

start_link/2
  slave, 257
  supervisor, 289
  supervisor_bridge, 294

start_link/3
  gen_fsm, 162
  gen_server, 171
  proc_lib, 225
  slave, 257
  supervisor, 289
  supervisor_bridge, 294

start_link/4
  gen_fsm, 162
  gen_server, 171
  proc_lib, 225

start_link/5
  proc_lib, 225
start_restricted/1
   shell, 254
start_timer/2
   gen_fsm, 165
statistics/2
   sys, 298
statistics/3
   sys, 298
stop/0
   pool, 222
stop/1
   gen_event, 157
      slave, 258
stop_restricted/1
   shell, 254
store/3
   dict, 78
str/2
   string, 282
strict_relation/1
   sofs, 278
string
   centre/2, 285
      centre/3, 285
   chars/2, 283
   chars/3, 283
   chr/2, 282
   concat/2, 282
   copies/2, 284
   cspan/2, 283
   equal/2, 282
   left/2, 285
   left/3, 285
   len/1, 282
   rchr/2, 282
   right/2, 285
   right/3, 285
   rstr/2, 282
   span/2, 283
   str/2, 282
   strip/1, 284
   strip/2, 284
   strip/3, 284
   sub_string/2, 285
   sub_string/3, 286
   sub_word/2, 284
   sub_word/3, 284
   substr/2, 283
   substr/3, 283
   tokens/2, 283
   words/1, 284
   words/2, 284
   string/1
      erl_scan, 108
   string/2
      erl_scan, 108
   strip/1
      beam_lib, 49
      string, 284
   strip/2
      string, 284
   strip/3
      string, 284
   strip_files/1
      beam_lib, 49
   strip_release/1
      beam_lib, 50
   strong_components/1
      digraph_utils, 90
   sub/3
      regexp, 240
   sub_keys/1
      win32reg, 311
   sub_string/2
      string, 285
   sub_string/3
      string, 286
   sub_word/2
      string, 284
   sub_word/3
      string, 284
   subgraph/3
      digraph_utils, 90
   sublist/2
      lists, 198
   sublist/3
      lists, 199
   substitute_aliases/2
      proplists, 232
   substitute_negations/2
      proplists, 232
substitution/2
sofs, 279
substr/2
string, 283
substr/3
string, 283
subtract/2
lists, 199
sets, 245
suffix/2
lists, 199
sum/1
lists, 199
supervisor
check_childspecs/1, 292
delete_child/2, 291
Module:init/1, 293
restart_child/2, 291
start_child/2, 289
start_link/2, 289
start_link/3, 289
terminate_child/2, 290
which_children/1, 292
supervisor_bridge
Module:init/1, 295
Module:terminate/2, 295
start_link/2, 294
start_link/3, 294
suspend/1
sys, 299
suspend/2
sys, 299
swap_handler/5
gen_event, 156
swap_sup_handler/5
gen_event, 157
symdiff/2
sofs, 279
symmetric_partition/2
sofs, 280
sync/1
dets, 73
sync_notify/2
gen_event, 154
sync_send_all_state_event/2
gen fsm, 164
sync_send_event/2
gen fsm, 163
sync_send_event/3
gen fsm, 163
sys
change_code/4, 299
change_code/5, 299
debug_options/1, 301
get_debug/3, 301
get_status/1, 299
get_status/2, 299
handle_debug/1, 301
handle_system_msg/6, 301
install/3, 300
install/4, 300
log/2, 298
log/3, 298
log_to_file/2, 298
log_to_file/3, 298
Mod:system_code_change/4, 302
Mod:system_continue/3, 302
Mod:system_terminate/4, 302
no_debug/1, 299
no_debug/2, 299
print_log/1, 302
remove/2, 300
remove/3, 300
resume/1, 299
resume/2, 299
statistics/2, 298
statistics/3, 298
suspend/1, 299
suspend/2, 299
trace/2, 299
trace/3, 299
t/1
erl_tar, 114
tab2file/2
ers, 130
tab2list/1
ers, 130
table/1
erl_tar, 114
table/2
erl_tar, 114
tail/1
queue, 235
take_largest/1
gb_sets, 147
gb_trees, 151
take_smallest/1
gb_sets, 147
gb_trees, 151
takewhile/2
lists, 203
tan/1
math, 206
tanh/1
math, 206
tc/3
timer, 306
terminate_child/2
supervisor, 290
test_ms/2
ets, 130
time_difference/2
calendar, 58
time_to_seconds/1
calendar, 59
timer
apply_after/4, 304
apply_interval/4, 305
cancel/1, 305
exit_after/2, 305
exit_after/3, 305
hms/3, 306
hours/1, 306
kill_after/1, 305
kill_after/2, 305
minutes/1, 306
now_diff/2, 306
seconds/1, 306
send_after/2, 304
send_after/3, 304
send_interval/2, 305
send_interval/3, 305
sleep/1, 305
start/0, 304
tc/3, 306
to_dets/2
ets, 131
to_ets/2
ets, 131
lists, 200
umerge/3
   lists, 200
umerge3/3
   lists, 200
unfold/1
   proplists, 232
uniform/0
   random, 237
uniform/1
   random, 237
uniform_s/1
   random, 238
uniform_s/2
   random, 238
union/1
   gb_sets, 146
   sets, 245
   sofs, 280
union/2
   gb_sets, 146
   sets, 245
   sofs, 280
union_of_family/1
   sofs, 281
universal_time/0
   calendar, 59
universal_time_to_local_time/2
   calendar, 59
update/3
   dict, 78
   gb_trees, 150
update/4
   dict, 79
update_counter/3
   dets, 74
   dict, 79
   ets, 131
update_counter/4
   ets, 131
update_counter/6
   ets, 131
usort/1
   lists, 200
usort/2
   lists, 200
valid_date/1
   calendar, 59
valid_date/3
   calendar, 59
value/2
   win32reg, 311
values/1
   gb_trees, 150
   win32reg, 311
version/1
   beam_lib, 48
vertex/2
   digraph, 86
vertices/1
   digraph, 86
weak_relation/1
   sofs, 281
which_children/1
   supervisor, 292
which_handlers/1
   gen_event, 157
wildcard/1
   filelib, 138
wildcard/2
   filelib, 138
win32reg
   change_key/2, 309
   change_key_create/2, 309
   close/1, 309
   current_key/1, 309
   delete_key/1, 309
   delete_value/2, 310
   expand/1, 310
   format_error/1, 310
   open/1, 310
   set_value/3, 310
   sub_keys/1, 311
   value/2, 311
   values/1, 311
words/1
   string, 284
words/2

STDLIB
339
string, 284
write/1
  io_lib, 189
write/2
  io, 181
  io_lib, 189
write_atom/1
  io_lib, 190
write_char/1
  io_lib, 190
write_string/1
  io_lib, 190

xm/1
  c, 54

zi/0
  c, 54