Erlang Run-Time System Application (ERTS)

Erlang Run-Time System Application (ERTS) 5.7.5
March 9 2011

March 9 2011

1 User's Guide

The Erlang Runtime System Application *ERTS*.

1.1 Match specifications in Erlang

A "match specification" (match_spec) is an Erlang term describing a small "program" that will try to match something (either the parameters to a function as used in the `erlang:trace_pattern/2` BIF, or the objects in an ETS table.). The match_spec in many ways works like a small function in Erlang, but is interpreted/compiled by the Erlang runtime system to something much more efficient than calling an Erlang function. The match_spec is also very limited compared to the expressiveness of real Erlang functions.

Match specifications are given to the BIF `erlang:trace_pattern/2` to execute matching of function arguments as well as to define some actions to be taken when the match succeeds (the `MatchBody` part). Match specifications can also be used in ETS, to specify objects to be returned from an `ets:select/2` call (or other select calls). The semantics and restrictions differ slightly when using match specifications for tracing and in ETS, the differences are defined in a separate paragraph below.

The most notable difference between a match_spec and an Erlang fun is of course the syntax. Match specifications are Erlang terms, not Erlang code. A match_spec also has a somewhat strange concept of exceptions. An exception (e.g., `badarg`) in the `MatchCondition` part, which resembles an Erlang guard, will generate immediate failure, while an exception in the `MatchBody` part, which resembles the body of an Erlang function, is implicitly caught and results in the single atom `'EXIT'`.

1.1.1 Grammar

A match_spec can be described in this *informal* grammar:

- MatchExpression ::= [MatchFunction, ...]
- MatchFunction ::= { MatchHead, MatchConditions, MatchBody }
- MatchHead ::= MatchVariable | `'_'` | [MatchHeadPart, ...]
- MatchHeadPart ::= term() | MatchVariable | `'_'`
- MatchVariable ::= '$<number>'
- MatchConditions ::= [MatchCondition, ...] | `[]`
- MatchCondition ::= { GuardFunction } | { GuardFunction, ConditionExpression, ... }
- BoolFunction ::= `is_atom` | `is_constant` | `is_float` | `is_integer` | `is_list` | `is_number` | `is_pid` | `is_port` | `is_reference` | `is_tuple` | `is_binary` | `is_function` | `is_record` | `is_seq_trace` | `'and'` | `'or'` | `'not'` | `'xor'` | `andalso` | `orelse`
- ConditionExpression ::= ExprMatchVariable | { GuardFunction } | { GuardFunction, ConditionExpression, ... } | TermConstruct
- ExprMatchVariable ::= MatchVariable (bound in the MatchHead) | `'$_'` | `'$$'`
- TermConstruct = {{}} | {{ ConditionExpression, ... }} | `[]` | [ConditionExpression, ...] | NonCompositeTerm | Constant
- NonCompositeTerm ::= term() (not list or tuple)
- Constant ::= {`const`, term()}

- GuardFunction ::= BoolFunction | `abs` | `element` | `hd` | `length` | `node` | `round` | `size` | `tl` | `trunc` | `'+'` | `'-'` | `'*'` | `'div'` | `'rem'` | `'band'` | `'bor'` | `'bxor'` | `'bnot'` | `'bsl'` | `'bsr'` | `'>'` | `'>='` | `'<'` | `'=<'` | `'=:='` | `'=='` | `'=/='` | `'/='` | `self` | `get_tcw`
- MatchBody ::= [ActionTerm]
- ActionTerm ::= ConditionExpression | ActionCall
- ActionCall ::= {ActionFunction} | {ActionFunction, ActionTerm, ...}
- ActionFunction ::= `set_seq_token` | `get_seq_token` | `message` | `return_trace` | `exception_trace` | `process_dump` | `enable_trace` | `disable_trace` | `trace` | `display` | `caller` | `set_tcw` | `silent`

1.1.2 Function descriptions

Functions allowed in all types of match specifications

The different functions allowed in `match_spec` work like this:

is_atom, is_constant, is_float, is_integer, is_list, is_number, is_pid, is_port, is_reference, is_tuple, is_binary, is_function: Like the corresponding guard tests in Erlang, return `true` or `false`.

is_record: Takes an additional parameter, which SHALL be the result of `record_info(size, <record_type>)`, like in `{is_record, '$1', rectype, record_info(size, rectype)}`.

'not': Negates its single argument (anything other than `false` gives `false`).

'and': Returns `true` if all its arguments (variable length argument list) evaluate to `true`, else `false`. Evaluation order is undefined.

'or': Returns `true` if any of its arguments evaluates to `true`. Variable length argument list. Evaluation order is undefined.

andalso: Like `'and'`, but quits evaluating its arguments as soon as one argument evaluates to something else than true. Arguments are evaluated left to right.

orelse: Like `'or'`, but quits evaluating as soon as one of its arguments evaluates to `true`. Arguments are evaluated left to right.

'xor': Only two arguments, of which one has to be true and the other false to return `true`; otherwise `'xor'` returns false.

abs, element, hd, length, node, round, size, tl, trunc, '+', '-', '', 'div', 'rem', 'band', 'bor', 'bxor', 'bnot', 'bsl', 'bsr', '>', '>=', '<', '=<', '=:=', '==', '=/=', '/=', self:* Work as the corresponding Erlang bif's (or operators). In case of bad arguments, the result depends on the context. In the `MatchConditions` part of the expression, the test fails immediately (like in an Erlang guard), but in the `MatchBody`, exceptions are implicitly caught and the call results in the atom `'EXIT'`.

Functions allowed only for tracing

is_seq_trace: Returns `true` if a sequential trace token is set for the current process, otherwise `false`.

set_seq_token: Works like `seq_trace:set_token/2`, but returns `true` on success and `'EXIT'` on error or bad argument. Only allowed in the `MatchBody` part and only allowed when tracing.

get_seq_token: Works just like `seq_trace:get_token/0`, and is only allowed in the `MatchBody` part when tracing.

message: Sets an additional message appended to the trace message sent. One can only set one additional message in the body; subsequent calls will replace the appended message. As a special case, `{message, false}` disables sending of trace messages ('call' and 'return_to') for this function call, just like if the match_spec had not matched, which can be useful if only the side effects of the `MatchBody` are desired. Another special case is `{message,`

`true}` which sets the default behavior, as if the function had no match_spec, trace message is sent with no extra information (if no other calls to `message` are placed before `{message, true}`, it is in fact a "noop").

Takes one argument, the message. Returns `true` and can only be used in the `MatchBody` part and when tracing.

return_trace: Causes a `return_from` trace message to be sent upon return from the current function. Takes no arguments, returns `true` and can only be used in the `MatchBody` part when tracing. If the process trace flag `silent` is active the `return_from` trace message is inhibited.

NOTE! If the traced function is tail recursive, this match spec function destroys that property. Hence, if a match spec executing this function is used on a perpetual server process, it may only be active for a limited time, or the emulator will eventually use all memory in the host machine and crash. If this match_spec function is inhibited using the `silent` process trace flag tail recursiveness still remains.

exception_trace: Same as *return_trace*, plus; if the traced function exits due to an exception, an `exception_from` trace message is generated, whether the exception is caught or not.

process_dump: Returns some textual information about the current process as a binary. Takes no arguments and is only allowed in the `MatchBody` part when tracing.

enable_trace: With one parameter this function turns on tracing like the Erlang call `erlang:trace(self(), true, [P2])`, where `P2` is the parameter to `enable_trace`. With two parameters, the first parameter should be either a process identifier or the registered name of a process. In this case tracing is turned on for the designated process in the same way as in the Erlang call `erlang:trace(P1, true, [P2])`, where P1 is the first and P2 is the second argument. The process `P1` gets its trace messages sent to the same tracer as the process executing the statement uses. `P1` can *not* be one of the atoms `all`, `new` or `existing` (unless, of course, they are registered names). `P2` can *not* be `cpu_timestamp` nor `{tracer,_}`. Returns `true` and may only be used in the `MatchBody` part when tracing.

disable_trace: With one parameter this function disables tracing like the Erlang call `erlang:trace(self(), false, [P2])`, where `P2` is the parameter to `disable_trace`. With two parameters it works like the Erlang call `erlang:trace(P1, false, [P2])`, where P1 can be either a process identifier or a registered name and is given as the first argument to the match_spec function. `P2` can *not* be `cpu_timestamp` nor `{tracer,_}`. Returns `true` and may only be used in the `MatchBody` part when tracing.

trace: With two parameters this function takes a list of trace flags to disable as first parameter and a list of trace flags to enable as second parameter. Logically, the disable list is applied first, but effectively all changes are applied atomically. The trace flags are the same as for `erlang:trace/3` not including `cpu_timestamp` but including `{tracer,_}`. If a tracer is specified in both lists, the tracer in the enable list takes precedence. If no tracer is specified the same tracer as the process executing the match spec is used. With three parameters to this function the first is either a process identifier or the registered name of a process to set trace flags on, the second is the disable list, and the third is the enable list. Returns `true` if any trace property was changed for the trace target process or `false` if not. It may only be used in the `MatchBody` part when tracing.

caller: Returns the calling function as a tuple {Module, Function, Arity} or the atom `undefined` if the calling function cannot be determined. May only be used in the `MatchBody` part when tracing.

Note that if a "technically built in function" (i.e. a function not written in Erlang) is traced, the `caller` function will sometimes return the atom `undefined`. The calling Erlang function is not available during such calls.

display: For debugging purposes only; displays the single argument as an Erlang term on stdout, which is seldom what is wanted. Returns `true` and may only be used in the `MatchBody` part when tracing.

get_tcw: Takes no argument and returns the value of the node's trace control word. The same is done by `erlang:system_info(trace_control_word)`.

The trace control word is a 32-bit unsigned integer intended for generic trace control. The trace control word can be tested and set both from within trace match specifications and with BIFs. This call is only allowed when tracing.

set_tcw: Takes one unsigned integer argument, sets the value of the node's trace control word to the value of the argument and returns the previous value. The same is done by `erlang:system_flag(trace_control_word, Value)`. It is only allowed to use `set_tcw` in the `MatchBody` part when tracing.

silent: Takes one argument. If the argument is `true`, the call trace message mode for the current process is set to silent for this call and all subsequent, i.e call trace messages are inhibited even if `{message, true}` is called in the `MatchBody` part for a traced function.

This mode can also be activated with the `silent` flag to `erlang:trace/3`.

If the argument is `false`, the call trace message mode for the current process is set to normal (non-silent) for this call and all subsequent.

If the argument is neither `true` nor `false`, the call trace message mode is unaffected.

Note that all "function calls" have to be tuples, even if they take no arguments. The value of `self` is the atom() `self`, but the value of `{self}` is the pid() of the current process.

1.1.3 Variables and literals

Variables take the form `'$<number>'` where `<number>` is an integer between 0 (zero) and 100000000 (1e+8), the behavior if the number is outside these limits is *undefined*. In the `MatchHead` part, the special variable `'_'` matches anything, and never gets bound (like _ in Erlang). In the `MatchCondition/MatchBody` parts, no unbound variables are allowed, why `'_'` is interpreted as itself (an atom). Variables can only be bound in the `MatchHead` part. In the `MatchBody` and `MatchCondition` parts, only variables bound previously may be used. As a special case, in the `MatchCondition/MatchBody` parts, the variable `'$_'` expands to the whole expression which matched the `MatchHead` (i.e., the whole parameter list to the possibly traced function or the whole matching object in the ets table) and the variable `'$$'` expands to a list of the values of all bound variables in order (i.e. `['$1','$2', ...]`).

In the `MatchHead` part, all literals (except the variables noted above) are interpreted as is. In the `MatchCondition/MatchBody` parts, however, the interpretation is in some ways different. Literals in the `MatchCondition/MatchBody` can either be written as is, which works for all literals except tuples, or by using the special form `{const, T}`, where `T` is any Erlang term. For tuple literals in the match_spec, one can also use double tuple parentheses, i.e., construct them as a tuple of arity one containing a single tuple, which is the one to be constructed. The "double tuple parenthesis" syntax is useful to construct tuples from already bound variables, like in `{{'$1', [a,b,'$2']}}`. Some examples may be needed:

Expression	Variable bindings	Result
{{'$1','$2'}}	'$1' = a, '$2' = b	{a,b}
{const, {'$1', '$2'}}	doesn't matter	{'$1', '$2'}
a	doesn't matter	a
'$1'	'$1' = []	[]
['$1']	'$1' = []	[[]]
[{{a}}]	doesn't matter	[{a}]
42	doesn't matter	42
"hello"	doesn't matter	"hello"

$1	doesn't matter	49 (the ASCII value for the character '1')

Table 1.1: Literals in the MatchCondition/MatchBody parts of a match_spec

1.1.4 Execution of the match

The execution of the match expression, when the runtime system decides whether a trace message should be sent, goes as follows:

For each tuple in the `MatchExpression` list and while no match has succeeded:

- Match the `MatchHead` part against the arguments to the function, binding the `'$<number>'` variables (much like in `ets:match/2`). If the `MatchHead` cannot match the arguments, the match fails.
- Evaluate each `MatchCondition` (where only `'$<number>'` variables previously bound in the `MatchHead` can occur) and expect it to return the atom `true`. As soon as a condition does not evaluate to `true`, the match fails. If any BIF call generates an exception, also fail.
- - *If the match_spec is executing when tracing:*
    Evaluate each `ActionTerm` in the same way as the `MatchConditions`, but completely ignore the return values. Regardless of what happens in this part, the match has succeeded.
  - *If the match_spec is executed when selecting objects from an ETS table:*
    Evaluate the expressions in order and return the value of the last expression (typically there is only one expression in this context)

1.1.5 Differences between match specifications in ETS and tracing

ETS match specifications are there to produce a return value. Usually the expression contains one single `ActionTerm` which defines the return value without having any side effects. Calls with side effects are not allowed in the ETS context.

When tracing there is no return value to produce, the match specification either matches or doesn't. The effect when the expression matches is a trace message rather then a returned term. The `ActionTerm`'s are executed as in an imperative language, i.e. for their side effects. Functions with side effects are also allowed when tracing.

In ETS the match head is a `tuple()` (or a single match variable) while it is a list (or a single match variable) when tracing.

1.1.6 Examples

Match an argument list of three where the first and third arguments are equal:

```
[{['$1', '_', '$1'],
  [],
  []}]
```

Match an argument list of three where the second argument is a number greater than three:

```
[{['_', '$1', '_'],
  [{ '>', '$1', 3}],
  []}]
```

Match an argument list of three, where the third argument is a tuple containing argument one and two *or* a list beginning with argument one and two (i. e. `[a,b,[a,b,c]]` or `[a,b,{a,b}]`):

```
[{['$1', '$2', '$3'],
  [{orelse,
      {'=:=', '$3', {{'$1','$2'}}},
      {'and',
        {'=:=', '$1', {hd, '$3'}},
        {'=:=', '$2', {hd, {tl, '$3'}}}}}}],
  []}]
```

The above problem may also be solved like this:

```
[{['$1', '$2', {'$1', '$2'}], [], []},
 {['$1', '$2', ['$1', '$2' | '_']], [], []}]
```

Match two arguments where the first is a tuple beginning with a list which in turn begins with the second argument times two (i. e. [{[4,x],y},2] or [{[8], y, z},4])

```
[{['$1', '$2'],[{'=:=', {'*', 2, '$2'}, {hd, {element, 1, '$1'}}}],
  []}]
```

Match three arguments. When all three are equal and are numbers, append the process dump to the trace message, else let the trace message be as is, but set the sequential trace token label to 4711.

```
[{['$1', '$1', '$1'],
  [{is_number, '$1'}],
  [{message, {process_dump}}]},
 {'_', [], [{set_seq_token, label, 4711}]}]
```

As can be noted above, the parameter list can be matched against a single `MatchVariable` or an `'_'`. To replace the whole parameter list with a single variable is a special case. In all other cases the `MatchHead` has to be a *proper* list.

Match all objects in an ets table where the first element is the atom 'strider' and the tuple arity is 3 and return the whole object.

```
[{{strider,'_'.'_'},
  [],
  ['$_']}]
```

Match all objects in an ets table with arity > 1 and the first element is 'gandalf', return element 2.

```
[{'$1',
  [{'==', gandalf, {element, 1, '$1'}},{'>=',{size, '$1'},2}],
```

```
  [{element,2,'$1'}]}]
```

In the above example, if the first element had been the key, it's much more efficient to match that key in the `MatchHead` part than in the `MatchConditions` part. The search space of the tables is restricted with regards to the `MatchHead` so that only objects with the matching key are searched.

Match tuples of 3 elements where the second element is either 'merry' or 'pippin', return the whole objects.

```
[{{'_',merry,'_'},
  [],
  ['$_']},
 {{'_',pippin,'_'},
  [],
  ['$_']}]
```

The function `ets:test_ms/2` can be useful for testing complicated ets matches.

1.2 How to interpret the Erlang crash dumps

This document describes the `erl_crash.dump` file generated upon abnormal exit of the Erlang runtime system.

Important: For OTP release R9C the Erlang crash dump has had a major facelift. This means that the information in this document will not be directly applicable for older dumps. However, if you use the Crashdump Viewer tool on older dumps, the crash dumps are translated into a format similar to this.

The system will write the crash dump in the current directory of the emulator or in the file pointed out by the environment variable (whatever that means on the current operating system) ERL_CRASH_DUMP. For a crash dump to be written, there has to be a writable file system mounted.

Crash dumps are written mainly for one of two reasons: either the builtin function `erlang:halt/1` is called explicitly with a string argument from running Erlang code, or else the runtime system has detected an error that cannot be handled. The most usual reason that the system can't handle the error is that the cause is external limitations, such as running out of memory. A crash dump due to an internal error may be caused by the system reaching limits in the emulator itself (like the number of atoms in the system, or too many simultaneous ets tables). Usually the emulator or the operating system can be reconfigured to avoid the crash, which is why interpreting the crash dump correctly is important.

The erlang crash dump is a readable text file, but it might not be very easy to read. Using the Crashdump Viewer tool in the `observer` application will simplify the task. This is an HTML based tool for browsing Erlang crash dumps.

1.2.1 General information

The first part of the dump shows the creation time for the dump, a slogan indicating the reason for the dump, the system version, of the node from which the dump originates, the compile time of the emulator running the originating node and the number of atoms in the atom table.

Reasons for crash dumps (slogan)

The reason for the dump is noted in the beginning of the file as *Slogan: <reason>* (the word "slogan" has historical roots). If the system is halted by the BIF `erlang:halt/1`, the slogan is the string parameter passed to the BIF, otherwise it is a description generated by the emulator or the (Erlang) kernel. Normally the message should be enough to understand the problem, but nevertheless some messages are described here. Note however that the suggested reasons for the crash are *only suggestions*. The exact reasons for the errors may vary depending on the local applications and the underlying operating system.

- "<*A*>: Cannot allocate <*N*> bytes of memory (of type "<*T*>")." - The system has run out of memory. <A> is the allocator that failed to allocate memory, <N> is the number of bytes that <A> tried to allocate, and <T> is the memory block type that the memory was needed for. The most common case is that a process stores huge amounts of data. In this case <T> is most often `heap`, `old_heap`, `heap_frag`, or `binary`. For more information on allocators see *erts_alloc(3)*.
- "<*A*>: Cannot reallocate <*N*> bytes of memory (of type "<*T*>")." - Same as above with the exception that memory was being reallocated instead of being allocated when the system ran out of memory.
- "Unexpected op code *N*" - Error in compiled code, `beam` file damaged or error in the compiler.
- "Module *Name* undefined" | "Function *Name* undefined" | "No function *Name*:*Name*/1" | "No function *Name*:start/2" - The kernel/stdlib applications are damaged or the start script is damaged.
- "Driver_select called with too large file descriptor `N`" - The number of file descriptors for sockets exceed 1024 (Unix only). The limit on file-descriptors in some Unix flavors can be set to over 1024, but only 1024 sockets/pipes can be used simultaneously by Erlang (due to limitations in the Unix `select` call). The number of open regular files is not affected by this.
- "Received SIGUSR1" - The SIGUSR1 signal was sent to the Erlang machine (Unix only).
- "Kernel pid terminated (*Who*) (*Exit-reason*)" - The kernel supervisor has detected a failure, usually that the `application_controller` has shut down (`Who` = `application_controller`, `Why` = `shutdown`). The application controller may have shut down for a number of reasons, the most usual being that the node name of the distributed Erlang node is already in use. A complete supervisor tree "crash" (i.e., the top supervisors have exited) will give about the same result. This message comes from the Erlang code and not from the virtual machine itself. It is always due to some kind of failure in an application, either within OTP or a "user-written" one. Looking at the error log for your application is probably the first step to take.
- "Init terminating in do_boot ()" - The primitive Erlang boot sequence was terminated, most probably because the boot script has errors or cannot be read. This is usually a configuration error - the system may have been started with a faulty `-boot` parameter or with a boot script from the wrong version of OTP.
- "Could not start kernel pid (*Who*) ()" - One of the kernel processes could not start. This is probably due to faulty arguments (like errors in a `-config` argument) or faulty configuration files. Check that all files are in their correct location and that the configuration files (if any) are not damaged. Usually there are also messages written to the controlling terminal and/or the error log explaining what's wrong.

Other errors than the ones mentioned above may occur, as the `erlang:halt/1` BIF may generate any message. If the message is not generated by the BIF and does not occur in the list above, it may be due to an error in the emulator. There may however be unusual messages that I haven't mentioned, that still are connected to an application failure. There is a lot more information available, so more thorough reading of the crash dump may reveal the crash reason. The size of processes, the number of ets tables and the Erlang data on each process stack can be useful for tracking down the problem.

Number of atoms

The number of atoms in the system at the time of the crash is shown as *Atoms: <number>*. Some ten thousands atoms is perfectly normal, but more could indicate that the BIF `erlang:list_to_atom/1` is used to dynamically generate a lot of *different* atoms, which is never a good idea.

1.2.2 Memory information

Under the tag *=memory* you will find information similar to what you can obtain on a living node with *erlang:memory()*.

1.2.3 Internal table information

The tags *=hash_table:<table_name>* and *=index_table:<table_name>* presents internal tables. These are mostly of interest for runtime system developers.

1.2.4 Allocated areas

Under the tag *=allocated_areas* you will find information similar to what you can obtain on a living node with *erlang:system_info(allocated_areas)*.

1.2.5 Allocator

Under the tag *=allocator:<A>* you will find various information about allocator <A>. The information is similar to what you can obtain on a living node with *erlang:system_info({allocator, <A>})*. For more information see the documentation of *erlang:system_info({allocator, <A>})*, and the *erts_alloc(3)* documentation.

1.2.6 Process information

The Erlang crashdump contains a listing of each living Erlang process in the system. The process information for one process may look like this (line numbers have been added):

The following fields can exist for a process:

=proc:<pid>
: Heading, states the process identifier

State
: The state of the process. This can be one of the following:

  - *Scheduled* - The process was scheduled to run but not currently running ("in the run queue").
  - *Waiting* - The process was waiting for something (in `receive`).
  - *Running* - The process was currently running. If the BIF `erlang:halt/1` was called, this was the process calling it.
  - *Exiting* - The process was on its way to exit.
  - *Garbing* - This is bad luck, the process was garbage collecting when the crash dump was written, the rest of the information for this process is limited.
  - *Suspended* - The process is suspended, either by the BIF `erlang:suspend_process/1` or because it is trying to write to a busy port.

Registered name
: The registered name of the process, if any.

Spawned as
: The entry point of the process, i.e., what function was referenced in the `spawn` or `spawn_link` call that started the process.

Last scheduled in for | Current call
: The current function of the process. These fields will not always exist.

Spawned by
: The parent of the process, i.e. the process which executed `spawn` or `spawn_link`.

Started
: The date and time when the process was started.

Message queue length
: The number of messages in the process' message queue.

Number of heap fragments
: The number of allocated heap fragments.

Heap fragment data
: Size of fragmented heap data. This is data either created by messages being sent to the process or by the Erlang BIFs. This amount depends on so many things that this field is utterly uninteresting.

Link list
: Process id's of processes linked to this one. May also contain ports. If process monitoring is used, this field also tells in which direction the monitoring is in effect, i.e., a link being "to" a process tells you that the "current"

process was monitoring the other and a link "from" a process tells you that the other process was monitoring the current one.

Reductions
The number of reductions consumed by the process.

Stack+heap
The size of the stack and heap (they share memory segment)

OldHeap
The size of the "old heap". The Erlang virtual machine uses generational garbage collection with two generations. There is one heap for new data items and one for the data that have survived two garbage collections. The assumption (which is almost always correct) is that data that survive two garbage collections can be "tenured" to a heap more seldom garbage collected, as they will live for a long period. This is a quite usual technique in virtual machines. The sum of the heaps and stack together constitute most of the process's allocated memory.

Heap unused, OldHeap unused
The amount of unused memory on each heap. This information is usually useless.

Stack
If the system uses shared heap, the fields *Stack+heap*, *OldHeap*, *Heap unused* and *OldHeap unused* do not exist. Instead this field presents the size of the process' stack.

Program counter
The current instruction pointer. This is only interesting for runtime system developers. The function into which the program counter points is the current function of the process.

CP
The continuation pointer, i.e. the return address for the current call. Usually useless for other than runtime system developers. This may be followed by the function into which the CP points, which is the function calling the current function.

Arity
The number of live argument registers. The argument registers, if any are live, will follow. These may contain the arguments of the function if they are not yet moved to the stack.

See also the section about *process data*.

1.2.7 Port information

This section lists the open ports, their owners, any linked processed, and the name of their driver or external process.

1.2.8 ETS tables

This section contains information about all the ETS tables in the system. The following fields are interesting for each table:

=ets:<owner>
Heading, states the owner of the table (a process identifier)

Table
The identifier for the table. If the table is a `named_table`, this is the name.

Name
The name of the table, regardless of whether it is a `named_table` or not.

Buckets
This occurs if the table is a hash table, i.e. if it is not an `ordered_set`.

Ordered set (AVL tree), Elements
This occurs only if the table is an `ordered_set`. (The number of elements is the same as the number of objects in the table.)

Objects
The number of objects in the table

Words
The number of words (usually 4 bytes/word) allocated to data in the table.

1.2.9 Timers

This section contains information about all the timers started with the BIFs `erlang:start_timer/3` and `erlang:send_after/3`. The following fields exists for each timer:

=timer:<owner>
Heading, states the owner of the timer (a process identifier) i.e. the process to receive the message when the timer expires.
Message
The message to be sent.
Time left
Number of milliseconds left until the message would have been sent.

1.2.10 Distribution information

If the Erlang node was alive, i.e., set up for communicating with other nodes, this section lists the connections that were active. The following fields can exist:

=node:<node_name>
The name of the node
no_distribution
This will only occur if the node was not distributed.
=visible_node:<channel>
Heading for a visible nodes, i.e. an alive node with a connection to the node that crashed. States the channel number for the node.
=hidden_node:<channel>
Heading for a hidden node. A hidden node is the same as a visible node, except that it is started with the "-hidden" flag. States the channel number for the node.
=not_connected:<channel>
Heading for a node which is has been connected to the crashed node earlier. References (i.e. process or port identifiers) to the not connected node existed at the time of the crash. exist. States the channel number for the node.
Name
The name of the remote node.
Controller
The port which controls the communication with the remote node.
Creation
An integer (1-3) which together with the node name identifies a specific instance of the node.
Remote monitoring: <local_proc> <remote_proc>
The local process was monitoring the remote process at the time of the crash.
Remotely monitored by: <local_proc> <remote_proc>
The remote process was monitoring the local process at the time of the crash.
Remote link: <local_proc> <remote_proc>
A link existed between the local process and the remote process at the time of the crash.

1.2.11 Loaded module information

This section contains information about all loaded modules. First, the memory usage by loaded code is summarized. There is one field for "Current code" which is code that is the current latest version of the modules. There is also a field for "Old code" which is code where there exists a newer version in the system, but the old version is not yet purged. The memory usage is in bytes.

All loaded modules are then listed. The following fields exist:

=mod:<module_name>
Heading, and the name of the module.

Current size
Memory usage for the loaded code in bytes

Old size
Memory usage for the old code, if any.

Current attributes
Module attributes for the current code. This field is decoded when looked at by the Crashdump Viewer tool.

Old attributes
Module attributes for the old code, if any. This field is decoded when looked at by the Crashdump Viewer tool.

Current compilation info
Compilation information (options) for the current code. This field is decoded when looked at by the Crashdump Viewer tool.

Old compilation info
Compilation information (options) for the old code, if any. This field is decoded when looked at by the Crashdump Viewer tool.

1.2.12 Fun information

In this section, all funs are listed. The following fields exist for each fun:

=fun
Heading

Module
The name of the module where the fun was defined.

Uniq, Index
Identifiers

Address
The address of the fun's code.

Native_address
The address of the fun's code when HiPE is enabled.

Refc
The number of references to the fun.

1.2.13 Process Data

For each process there will be at least one *=proc_stack* and one *=proc_heap* tag followed by the raw memory information for the stack and heap of the process.

For each process there will also be a *=proc_messages* tag if the process' message queue is non-empty and a *=proc_dictionary* tag if the process' dictionary (the `put/2` and `get/1` thing) is non-empty.

The raw memory information can be decoded by the Crashdump Viewer tool. You will then be able to see the stack dump, the message queue (if any) and the dictionary (if any).

The stack dump is a dump of the Erlang process stack. Most of the live data (i.e., variables currently in use) are placed on the stack; thus this can be quite interesting. One has to "guess" what's what, but as the information is symbolic, thorough reading of this information can be very useful. As an example we can find the state variable of the Erlang primitive loader on line `(5)` in the example below:

```
(1)  3cac44   Return addr 0x13BF58 (<terminate process normally>)
(2)  y(0)     ["/view/siri_r10_dev/clearcase/otp/erts/lib/kernel/ebin","/view/siri_r10_dev/
(3)  clearcase/otp/erts/lib/stdlib/ebin"]
```

```
(4)  y(1)     <0.1.0>
(5)  y(2)     {state,[],none,#Fun<erl_prim_loader.6.7085890>,undefined,#Fun<erl_prim_loader.7.9000327>,#Fun<erl_
(6)  y(3)     infinity
```

When interpreting the data for a process, it is helpful to know that anonymous function objects (funs) are given a name constructed from the name of the function in which they are created, and a number (starting with 0) indicating the number of that fun within that function.

1.2.14 Atoms

Now all the atoms in the system are written. This is only interesting if one suspects that dynamic generation of atoms could be a problem, otherwise this section can be ignored.

Note that the last created atom is printed first.

1.2.15 Disclaimer

The format of the crash dump evolves between releases of OTP. Some information here may not apply to your version. A description as this will never be complete; it is meant as an explanation of the crash dump in general and as a help when trying to find application errors, not as a complete specification.

1.3 How to implement an alternative carrier for the Erlang distribution

This document describes how one can implement ones own carrier protocol for the Erlang distribution. The distribution is normally carried by the TCP/IP protocol. What's explained here is the method for replacing TCP/IP with another protocol.

The document is a step by step explanation of the `uds_dist` example application (seated in the kernel applications `examples` directory). The `uds_dist` application implements distribution over Unix domain sockets and is written for the Sun Solaris 2 operating environment. The mechanisms are however general and applies to any operating system Erlang runs on. The reason the C code is not made portable, is simply readability.

> **Note:**
>
> This document was written a long time ago. Most of it is still valid, but some things have changed since it was first written. Most notably the driver interface. There have been some updates to the documentation of the driver presented in this documentation, but more could be done and are planned for the future. The reader is encouraged to also read the *erl_driver*, and the *driver_entry* documentation.

1.3.1 Introduction

To implement a new carrier for the Erlang distribution, one must first make the protocol available to the Erlang machine, which involves writing an Erlang driver. There is no way one can use a port program, there *has* to be an Erlang driver. Erlang drivers can either be statically linked to the emulator, which can be an alternative when using the open source distribution of Erlang, or dynamically loaded into the Erlang machines address space, which is the only alternative if a precompiled version of Erlang is to be used.

Writing an Erlang driver is by no means easy. The driver is written as a couple of call-back functions called by the Erlang emulator when data is sent to the driver or the driver has any data available on a file descriptor. As the driver call-back routines execute in the main thread of the Erlang machine, the call-back functions can perform no blocking activity whatsoever. The call-backs should only set up file descriptors for waiting and/or read/write available data. All I/O has to be non blocking. Driver call-backs are however executed in sequence, why a global state can safely be updated within the routines.

When the driver is implemented, one would preferably write an Erlang interface for the driver to be able to test the functionality of the driver separately. This interface can then be used by the distribution module which will cover the details of the protocol from the `net_kernel`. The easiest path is to mimic the `inet` and `inet_tcp` interfaces, but a lot of functionality in those modules need not be implemented. In the example application, only a few of the usual interfaces are implemented, and they are much simplified.

When the protocol is available to Erlang through a driver and an Erlang interface module, a distribution module can be written. The distribution module is a module with well defined call-backs, much like a `gen_server` (there is no compiler support for checking the call-backs though). The details of finding other nodes (i.e. talking to epmd or something similar), creating a listen port (or similar), connecting to other nodes and performing the handshakes/cookie verification are all implemented by this module. There is however a utility module, `dist_util`, that will do most of the hard work of handling handshakes, cookies, timers and ticking. Using `dist_util` makes implementing a distribution module much easier and that's what we are doing in the example application.

The last step is to create boot scripts to make the protocol implementation available at boot time. The implementation can be debugged by starting the distribution when all of the system is running, but in a real system the distribution should start very early, why a boot-script and some command line parameters are necessary. This last step also implies that the Erlang code in the interface and distribution modules is written in such a way that it can be run in the startup phase. Most notably there can be no calls to the `application` module or to any modules not loaded at boot-time (i.e. only `kernel`, `stdlib` and the application itself can be used).

1.3.2 The driver

Although Erlang drivers in general may be beyond the scope of this document, a brief introduction seems to be in place.

Drivers in general

An Erlang driver is a native code module written in C (or assembler) which serves as an interface for some special operating system service. This is a general mechanism that is used throughout the Erlang emulator for all kinds of I/O. An Erlang driver can be dynamically linked (or loaded) to the Erlang emulator at runtime by using the `erl_ddll` Erlang module. Some of the drivers in OTP are however statically linked to the runtime system, but that's more an optimization than a necessity.

The driver data-types and the functions available to the driver writer are defined in the header file `erl_driver.h` (there is also an deprecated version called `driver.h`, don't use that one.) seated in Erlang's include directory (and in $ERL_TOP/erts/emulator/beam in the source code distribution). Refer to that file for function prototypes etc.

When writing a driver to make a communications protocol available to Erlang, one should know just about everything worth knowing about that particular protocol. All operation has to be non blocking and all possible situations should be accounted for in the driver. A non stable driver will affect and/or crash the whole Erlang runtime system, which is seldom what's wanted.

The emulator calls the driver in the following situations:

- When the driver is loaded. This call-back has to have a special name and will inform the emulator of what call-backs should be used by returning a pointer to a `ErlDrvEntry` struct, which should be properly filled in (see below).
- When a port to the driver is opened (by a `open_port` call from Erlang). This routine should set up internal data structures and return an opaque data entity of the type `ErlDrvData`, which is a data-type large enough to hold a pointer. The pointer returned by this function will be the first argument to all other call-backs concerning this particular port. It is usually called the port handle. The emulator only stores the handle and does never try to interpret it, why it can be virtually anything (well anything not larger than a pointer that is) and can point to anything if it is a pointer. Usually this pointer will refer to a structure holding information about the particular port, as i t does in our example.
- When an Erlang process sends data to the port. The data will arrive as a buffer of bytes, the interpretation is not defined, but is up to the implementor. This call-back returns nothing to the caller, answers are sent to the caller

as messages (using a routine called `driver_output` available to all drivers). There is also a way to talk in a synchronous way to drivers, described below. There can be an additional call-back function for handling data that is fragmented (sent in a deep io-list). That interface will get the data in a form suitable for Unix `writev` rather than in a single buffer. There is no need for a distribution driver to implement such a call-back, so we wont.

- When a file descriptor is signaled for input. This call-back is called when the emulator detects input on a file descriptor which the driver has marked for monitoring by using the interface `driver_select`. The mechanism of driver select makes it possible to read non blocking from file descriptors by calling `driver_select` when reading is needed and then do the actual reading in this call-back (when reading is actually possible). The typical scenario is that `driver_select` is called when an Erlang process orders a read operation, and that this routine sends the answer when data is available on the file descriptor.
- When a file descriptor is signaled for output. This call-back is called in a similar way as the previous, but when writing to a file descriptor is possible. The usual scenario is that Erlang orders writing on a file descriptor and that the driver calls `driver_select`. When the descriptor is ready for output, this call-back is called an the driver can try to send the output. There may of course be queuing involved in such operations, and there are some convenient queue routines available to the driver writer to use in such situations.
- When a port is closed, either by an Erlang process or by the driver calling one of the `driver_failure_XXX` routines. This routine should clean up everything connected to one particular port. Note that when other call-backs call a `driver_failure_XXX` routine, this routine will be immediately called and the call-back routine issuing the error can make no more use of the data structures for the port, as this routine surely has freed all associated data and closed all file descriptors. If the queue utility available to driver writes is used, this routine will however *not* be called until the queue is empty.
- When an Erlang process calls `erlang:port_control/3`, which is a synchronous interface to drivers. The control interface is used to set driver options, change states of ports etc. We'll use this interface quite a lot in our example.
- When a timer expires. The driver can set timers with the function `driver_set_timer`. When such timers expire, a specific call-back function is called. We will not use timers in our example.
- When the whole driver is unloaded. Every resource allocated by the driver should be freed.

The distribution driver's data structures

The driver used for Erlang distribution should implement a reliable, order maintaining, variable length packet oriented protocol. All error correction, re-sending and such need to be implemented in the driver or by the underlying communications protocol. If the protocol is stream oriented (as is the case with both TCP/IP and our streamed Unix domain sockets), some mechanism for packaging is needed. We will use the simple method of having a header of four bytes containing the length of the package in a big endian 32 bit integer (as Unix domain sockets only can be used between processes on the same machine, we actually don't need to code the integer in some special endianess, but I'll do it anyway because in most situation you do need to do it. Unix domain sockets are reliable and order maintaining, so we don't need to implement resends and such in our driver.

Lets start writing our example Unix domain sockets driver by declaring prototypes and filling in a static ErlDrvEntry structure.

```
( 1) #include <stdio.h>
( 2) #include <stdlib.h>
( 3) #include <string.h>
( 4) #include <unistd.h>
( 5) #include <errno.h>
( 6) #include <sys/types.h>
( 7) #include <sys/stat.h>
( 8) #include <sys/socket.h>
( 9) #include <sys/un.h>
(10) #include <fcntl.h>
```

```
(11) #define HAVE_UIO_H
(12) #include "erl_driver.h"

(13) /*
(14) ** Interface routines
(15) */
(16) static ErlDrvData uds_start(ErlDrvPort port, char *buff);
(17) static void uds_stop(ErlDrvData handle);
(18) static void uds_command(ErlDrvData handle, char *buff, int bufflen);
(19) static void uds_input(ErlDrvData handle, ErlDrvEvent event);
(20) static void uds_output(ErlDrvData handle, ErlDrvEvent event);
(21) static void uds_finish(void);
(22) static int uds_control(ErlDrvData handle, unsigned int command,
(23)                       char* buf, int count, char** res, int res_size);

(24) /* The driver entry */
(25) static ErlDrvEntry uds_driver_entry = {
(26)     NULL,                           /* init, N/A */
(27)     uds_start,                      /* start, called when port is opened */
(28)     uds_stop,                       /* stop, called when port is closed */
(29)     uds_command,                    /* output, called when erlang has sent */
(30)     uds_input,                      /* ready_input, called when input
(31)                                        descriptor ready */
(32)     uds_output,                     /* ready_output, called when output
(33)                                        descriptor ready */
(34)     "uds_drv",                      /* char *driver_name, the argument
(35)                                        to open_port */
(36)     uds_finish,                     /* finish, called when unloaded */
(37)     NULL,                           /* void * that is not used (BC) */
(38)     uds_control,                    /* control, port_control callback */
(39)     NULL,                           /* timeout, called on timeouts */
(40)     NULL,                           /* outputv, vector output interface */
(41)     NULL,                           /* ready_async callback */
(42)     NULL,                           /* flush callback */
(43)     NULL,                           /* call callback */
(44)     NULL,                           /* event callback */
(45)     ERL_DRV_EXTENDED_MARKER,        /* Extended driver interface marker */
(46)     ERL_DRV_EXTENDED_MAJOR_VERSION, /* Major version number */
(47)     ERL_DRV_EXTENDED_MINOR_VERSION, /* Minor version number */
(48)     ERL_DRV_FLAG_SOFT_BUSY,         /* Driver flags. Soft busy flag is
(49)                                        required for distribution drivers */
(50)     NULL,                           /* Reserved for internal use */
(51)     NULL,                           /* process_exit callback */
(52)     NULL                            /* stop_select callback */
(53) };
```

On line 1 to 10 we have included the OS headers needed for our driver. As this driver is written for Solaris, we know that the header `uio.h` exists, why we can define the preprocessor variable `HAVE_UIO_H` before we include `erl_driver.h` at line 12. The definition of `HAVE_UIO_H` will make the I/O vectors used in Erlang's driver queues to correspond to the operating systems ditto, which is very convenient.

The different call-back functions are declared ("forward declarations") on line 16 to 23.

The driver structure is similar for statically linked in drivers and dynamically loaded. However some of the fields should be left empty (i.e. initialized to NULL) in the different types of drivers. The first field (the `init` function pointer) is always left blank in a dynamically loaded driver, which can be seen on line 26. The NULL on line 37 should always be there, the field is no longer used and is retained for backward compatibility. We use no timers in this driver, why no call-back for timers is needed. The `outputv` field (line 40) can be used to implement an interface similar to Unix `writev` for output. The Erlang runtime system could previously not use `outputv` for the distribution, but since erts version 5.7.2 it can. Since this driver was written before erts version 5.7.2 it does not use the `outputv`

callback. Using the `outputv` callback is preferred since it reduces copying of data. (We will however use scatter/gather I/O internally in the driver).

As of erts version 5.5.3 the driver interface was extended with version control and the possibility to pass capability information. Capability flags are present at line 48. As of erts version 5.7.4 the *ERL_DRV_FLAG_SOFT_BUSY* flag is required for drivers that are to be used by the distribution. The soft busy flag implies that the driver is capable of handling calls to the `output` and `outputv` callbacks even though it has marked itself as busy. This has always been a requirement on drivers used by the distribution, but there have previously not been any capability information available about this. For more information see *set_busy_port()*).

This driver was written before the runtime system had SMP support. The driver will still function in the runtime system with SMP support, but performance will suffer from lock contention on the driver lock used for the driver. This can be alleviated by reviewing and perhaps rewriting the code so that each instance of the driver safely can execute in parallel. When instances safely can execute in parallel it is safe to enable instance specific locking on the driver. This is done by passing *ERL_DRV_FLAG_USE_PORT_LOCKING* as a driver flag. This is left as an exercise for the reader.

Our defined call-backs thus are:

- uds_start, which shall initiate data for a port. We wont create any actual sockets here, just initialize data structures.
- uds_stop, the function called when a port is closed.
- uds_command, which will handle messages from Erlang. The messages can either be plain data to be sent or more subtle instructions to the driver. We will use this function mostly for data pumping.
- uds_input, this is the call-back which is called when we have something to read from a socket.
- uds_output, this is the function called when we can write to a socket.
- uds_finish, which is called when the driver is unloaded. A distribution driver will actually (or hopefully) never be unloaded, but we include this for completeness. Being able to clean up after oneself is always a good thing.
- uds_control, the `erlang:port_control/2` call-back, which will be used a lot in this implementation.

The ports implemented by this driver will operate in two major modes, which i will call the *command* and *data* modes. In command mode, only passive reading and writing (like gen_tcp:recv/gen_tcp:send) can be done, and this is the mode the port will be in during the distribution handshake. When the connection is up, the port will be switched to data mode and all data will be immediately read and passed further to the Erlang emulator. In data mode, no data arriving to the uds_command will be interpreted, but just packaged and sent out on the socket. The uds_control call-back will do the switching between those two modes.

While the `net_kernel` informs different subsystems that the connection is coming up, the port should accept data to send, but not receive any data, to avoid that data arrives from another node before every kernel subsystem is prepared to handle it. We have a third mode for this intermediate stage, lets call it the *intermediate* mode.

Lets define an enum for the different types of ports we have:

```
( 1) typedef enum {
( 2)     portTypeUnknown,      /* An uninitialized port */
( 3)     portTypeListener,     /* A listening port/socket */
( 4)     portTypeAcceptor,     /* An intermidiate stage when accepting
( 5)                              on a listen port */
( 6)     portTypeConnector,    /* An intermediate stage when connecting */
( 7)     portTypeCommand,      /* A connected open port in command mode */
( 8)     portTypeIntermediate, /* A connected open port in special
( 9)                              half active mode */
(10)     portTypeData          /* A connectec open port in data mode */
(11) } PortType;
```

Lets look at the different types:

- portTypeUnknown - The type a port has when it's opened, but not actually bound to any file descriptor.
- portTypeListener - A port that is connected to a listen socket. This port will not do especially much, there will be no data pumping done on this socket, but there will be read data available when one is trying to do an accept on the port.
- portTypeAcceptor - This is a port that is to represent the result of an accept operation. It is created when one wants to accept from a listen socket, and it will be converted to a portTypeCommand when the accept succeeds.
- portTypeConnector - Very similar to portTypeAcceptor, an intermediate stage between the request for a connect operation and that the socket is really connected to an accepting ditto in the other end. As soon as the sockets are connected, the port will switch type to portTypeCommand.
- portTypeCommand - A connected socket (or accepted socket if you want) that is in the command mode mentioned earlier.
- portTypeIntermediate - The intermediate stage for a connected socket. There should be no processing of input for this socket.
- portTypeData - The mode where data is pumped through the port and the uds_command routine will regard every call as a call where sending is wanted. In this mode all input available will be read and sent to Erlang as soon as it arrives on the socket, much like in the active mode of a `gen_tcp` socket.

Now lets look at the state we'll need for our ports. One can note that not all fields are used for all types of ports and that one could save some space by using unions, but that would clutter the code with multiple indirections, so i simply use one struct for all types of ports, for readability.

```
( 1) typedef unsigned char Byte;
( 2) typedef unsigned int Word;

( 3) typedef struct uds_data {
( 4)     int fd;                   /* File descriptor */
( 5)     ErlDrvPort port;          /* The port identifier */
( 6)     int lockfd;               /* The file descriptor for a lock file in
( 7)                                  case of listen sockets */
( 8)     Byte creation;            /* The creation serial derived from the
( 9)                                  lockfile */
(10)     PortType type;            /* Type of port */
(11)     char *name;               /* Short name of socket for unlink */
(12)     Word sent;                /* Bytes sent */
(13)     Word received;            /* Bytes received */
(14)     struct uds_data *partner; /* The partner in an accept/listen pair */
(15)     struct uds_data *next;    /* Next structure in list */
(16)     /* The input buffer and it's data */
(17)     int buffer_size;          /* The allocated size of the input buffer */
(18)     int buffer_pos;           /* Current position in input buffer */
(19)     int header_pos;           /* Where the current header is in the
(20)                                  input buffer */
(21)     Byte *buffer;             /* The actual input buffer */
(22) } UdsData;
```

This structure is used for all types of ports although some fields are useless for some types. The least memory consuming solution would be to arrange this structure as a union of structures, but the multiple indirections in the code to access a field in such a structure will clutter the code to much for an example.

Let's look at the fields in our structure:

- fd - The file descriptor of the socket associated with the port.
- port - The port identifier for the port which this structure corresponds to. It is needed for most `driver_XXX` calls from the driver back to the emulator.
- lockfd - If the socket is a listen socket, we use a separate (regular) file for two purposes:

  - We want a locking mechanism that gives no race conditions, so that we can be sure of if another Erlang node uses the listen socket name we require or if the file is only left there from a previous (crashed) session.
  - We store the *creation* serial number in the file. The *creation* is a number that should change between different instances of different Erlang emulators with the same name, so that process identifiers from one emulator won't be valid when sent to a new emulator with the same distribution name. The creation can be between 0 and 3 (two bits) and is stored in every process identifier sent to another node.

    In a system with TCP based distribution, this data is kept in the *Erlang port mapper daemon* (`epmd`), which is contacted when a distributed node starts. The lock-file and a convention for the UDS listen socket's name will remove the need for `epmd` when using this distribution module. UDS is always restricted to one host, why avoiding a port mapper is easy.
- creation - The creation number for a listen socket, which is calculated as (the value found in the lock-file + 1) rem 4. This creation value is also written back into the lock-file, so that the next invocation of the emulator will found our value in the file.
- type - The current type/state of the port, which can be one of the values declared above.
- name - The name of the socket file (the path prefix removed), which allows for deletion (`unlink`) when the socket is closed.
- sent - How many bytes that have been sent over the socket. This may wrap, but that's no problem for the distribution, as the only thing that interests the Erlang distribution is if this value has changed (the Erlang net_kernel *ticker* uses this value by calling the driver to fetch it, which is done through the `erlang:port_control` routine).
- received - How many bytes that are read (received) from the socket, used in similar ways as `sent`.
- partner - A pointer to another port structure, which is either the listen port from which this port is accepting a connection or the other way around. The "partner relation" is always bidirectional.
- next - Pointer to next structure in a linked list of all port structures. This list is used when accepting connections and when the driver is unloaded.
- buffer_size, buffer_pos, header_pos, buffer - data for input buffering. Refer to the source code (in the kernel/examples directory) for details about the input buffering. That certainly goes beyond the scope of this document.

Selected parts of the distribution driver implementation

The distribution drivers implementation is not completely covered in this text, details about buffering and other things unrelated to driver writing are not explained. Likewise are some peculiarities of the UDS protocol not explained in detail. The chosen protocol is not important.

Prototypes for the driver call-back routines can be found in the `erl_driver.h` header file.

The driver initialization routine is (usually) declared with a macro to make the driver easier to port between different operating systems (and flavours of systems). This is the only routine that has to have a well defined name. All other call-backs are reached through the driver structure. The macro to use is named `DRIVER_INIT` and takes the driver name as parameter.

```
(1) /* Beginning of linked list of ports */
(2) static UdsData *first_data;


(3) DRIVER_INIT(uds_drv)
(4) {
(5)     first_data = NULL;
(6)     return &uds_driver_entry;
```

```
(7) }
```

The routine initializes the single global data structure and returns a pointer to the driver entry. The routine will be called when `erl_ddll:load_driver` is called from Erlang.

The `uds_start` routine is called when a port is opened from Erlang. In our case, we only allocate a structure and initialize it. Creating the actual socket is left to the `uds_command` routine.

```
( 1) static ErlDrvData uds_start(ErlDrvPort port, char *buff)
( 2) {
( 3)     UdsData *ud;
( 4)
( 5)     ud = ALLOC(sizeof(UdsData));
( 6)     ud->fd = -1;
( 7)     ud->lockfd = -1;
( 8)     ud->creation = 0;
( 9)     ud->port = port;
(10)     ud->type = portTypeUnknown;
(11)     ud->name = NULL;
(12)     ud->buffer_size = 0;
(13)     ud->buffer_pos = 0;
(14)     ud->header_pos = 0;
(15)     ud->buffer = NULL;
(16)     ud->sent = 0;
(17)     ud->received = 0;
(18)     ud->partner = NULL;
(19)     ud->next = first_data;
(20)     first_data = ud;
(21)
(22)     return((ErlDrvData) ud);
(23) }
```

Every data item is initialized, so that no problems will arise when a newly created port is closed (without there being any corresponding socket). This routine is called when `open_port({spawn, "uds_drv"},[])` is called from Erlang.

The `uds_command` routine is the routine called when an Erlang process sends data to the port. All asynchronous commands when the port is in *command mode* as well as the sending of all data when the port is in *data mode* is handled in this9s routine. Let's have a look at it:

```
( 1) static void uds_command(ErlDrvData handle, char *buff, int bufflen)
( 2) {
( 3)     UdsData *ud = (UdsData *) handle;

( 4)     if (ud->type == portTypeData || ud->type == portTypeIntermediate) {
( 5)         DEBUGF(("Passive do_send %d",bufflen));
( 6)         do_send(ud, buff + 1, bufflen - 1); /* XXX */
( 7)         return;
( 8)     }
( 9)     if (bufflen == 0) {
(10)         return;
(11)     }
(12)     switch (*buff) {
(13)     case 'L':
(14)         if (ud->type != portTypeUnknown) {
(15)             driver_failure_posix(ud->port, ENOTSUP);
(16)             return;
(17)         }
(18)         uds_command_listen(ud,buff,bufflen);
```

```
(19)         return;
(20)     case 'A':
(21)         if (ud->type != portTypeUnknown) {
(22)             driver_failure_posix(ud->port, ENOTSUP);
(23)             return;
(24)         }
(25)         uds_command_accept(ud,buff,bufflen);
(26)         return;
(27)     case 'C':
(28)         if (ud->type != portTypeUnknown) {
(29)             driver_failure_posix(ud->port, ENOTSUP);
(30)             return;
(31)         }
(32)         uds_command_connect(ud,buff,bufflen);
(33)         return;
(34)     case 'S':
(35)         if (ud->type != portTypeCommand) {
(36)             driver_failure_posix(ud->port, ENOTSUP);
(37)             return;
(38)         }
(39)         do_send(ud, buff + 1, bufflen - 1);
(40)         return;
(41)     case 'R':
(42)         if (ud->type != portTypeCommand) {
(43)             driver_failure_posix(ud->port, ENOTSUP);
(44)             return;
(45)         }
(46)         do_recv(ud);
(47)         return;
(48)     default:
(49)         return;
(50)     }
(51) }
```

The command routine takes three parameters; the handle returned for the port by `uds_start`, which is a pointer to the internal port structure, the data buffer and the length of the data buffer. The buffer is the data sent from Erlang (a list of bytes) converted to an C array (of bytes).

If Erlang sends i.e. the list `[$a,$b,$c]` to the port, the `bufflen` variable will be `3` ant the `buff` variable will contain `{'a','b','c'}` (no null termination). Usually the first byte is used as an opcode, which is the case in our driver to (at least when the port is in command mode). The opcodes are defined as:

- 'L'<socketname>: Create and listen on socket with the given name.
- 'A'<listennumber as 32 bit bigendian>: Accept from the listen socket identified by the given identification number. The identification number is retrieved with the uds_control routine.
- 'C'<socketname>: Connect to the socket named <socketname>.
- 'S'<data>: Send the data <data> on the connected/accepted socket (in command mode). The sending is acked when the data has left this process.
- 'R': Receive one packet of data.

One may wonder what is meant by "one packet of data" in the 'R' command. This driver always sends data packeted with a 4 byte header containing a big endian 32 bit integer that represents the length of the data in the packet. There is no need for different packet sizes or some kind of streamed mode, as this driver is for the distribution only. One may wonder why the header word is coded explicitly in big endian when an UDS socket is local to the host. The answer simply is that I see it as a good practice when writing a distribution driver, as distribution in practice usually cross the host boundaries.

On line 4-8 we handle the case where the port is in data or intermediate mode, the rest of the routine handles the different commands. We see (first on line 15) that the routine uses the `driver_failure_posix()` routine to report errors. One important thing to remember is that the failure routines make a call to our `uds_stop` routine, which will remove

the internal port data. The handle (and the casted handle `ud`) is therefore *invalid pointers* after a `driver_failure` call and we should *immediately return*. The runtime system will send exit signals to all linked processes.

The uds_input routine gets called when data is available on a file descriptor previously passed to the `driver_select` routine. Typically this happens when a read command is issued and no data is available. Lets look at the `do_recv` routine:

```
( 1) static void do_recv(UdsData *ud)
( 2) {
( 3)     int res;
( 4)     char *ibuf;
( 5)     for(;;) {
( 6)         if ((res = buffered_read_package(ud,&ibuf)) < 0) {
( 7)             if (res == NORMAL_READ_FAILURE) {
( 8)                 driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 1);
( 9)             } else {
(10)                 driver_failure_eof(ud->port);
(11)             }
(12)             return;
(13)         }
(14)         /* Got a package */
(15)         if (ud->type == portTypeCommand) {
(16)             ibuf[-1] = 'R'; /* There is always room for a single byte
(17)                                opcode before the actual buffer
(18)                                (where the packet header was) */
(19)             driver_output(ud->port,ibuf - 1, res + 1);
(20)             driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ,0);
(21)             return;
(22)         } else {
(23)             ibuf[-1] = DIST_MAGIC_RECV_TAG; /* XXX */
(24)             driver_output(ud->port,ibuf - 1, res + 1);
(25)             driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ,1);
(26)         }
(27)     }
(28) }
```

The routine tries to read data until a packet is read or the `buffered_read_package` routine returns a `NORMAL_READ_FAILURE` (an internally defined constant for the module that means that the read operation resulted in an `EWOULDBLOCK`). If the port is in command mode, the reading stops when one package is read, but if it is in data mode, the reading continues until the socket buffer is empty (read failure). If no more data can be read and more is wanted (always the case when socket is in data mode) driver_select is called to make the `uds_input` call-back be called when more data is available for reading.

When the port is in data mode, all data is sent to Erlang in a format that suits the distribution, in fact the raw data will never reach any Erlang process, but will be translated/interpreted by the emulator itself and then delivered in the correct format to the correct processes. In the current emulator version, received data should be tagged with a single byte of 100. Thats what the macro `DIST_MAGIC_RECV_TAG` is defined to. The tagging of data in the distribution will possibly change in the future.

The `uds_input` routine will handle other input events (like nonblocking `accept`), but most importantly handle data arriving at the socket by calling `do_recv`:

```
( 1) static void uds_input(ErlDrvData handle, ErlDrvEvent event)
( 2) {
( 3)     UdsData *ud = (UdsData *) handle;

( 4)     if (ud->type == portTypeListener) {
( 5)         UdsData *ad = ud->partner;
```

```
( 6)         struct sockaddr_un peer;
( 7)         int pl = sizeof(struct sockaddr_un);
( 8)         int fd;

( 9)         if ((fd = accept(ud->fd, (struct sockaddr *) &peer, &pl)) < 0) {
(10)             if (errno != EWOULDBLOCK) {
(11)                 driver_failure_posix(ud->port, errno);
(12)                 return;
(13)             }
(14)             return;
(15)         }
(16)         SET_NONBLOCKING(fd);
(17)         ad->fd = fd;
(18)         ad->partner = NULL;
(19)         ad->type = portTypeCommand;
(20)         ud->partner = NULL;
(21)         driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
(22)         driver_output(ad->port, "Aok",3);
(23)         return;
(24)     }
(25)     do_recv(ud);
(26) }
```

The important line here is the last line in the function, the `do_read` routine is called to handle new input. The rest of the function handles input on a listen socket, which means that there should be possible to do an accept on the socket, which is also recognized as a read event.

The output mechanisms are similar to the input. Lets first look at the `do_send` routine:

```
( 1) static void do_send(UdsData *ud, char *buff, int bufflen)
( 2) {
( 3)     char header[4];
( 4)     int written;
( 5)     SysIOVec iov[2];
( 6)     ErlIOVec eio;
( 7)     ErlDrvBinary *binv[] = {NULL,NULL};

( 8)     put_packet_length(header, bufflen);
( 9)     iov[0].iov_base = (char *) header;
(10)     iov[0].iov_len = 4;
(11)     iov[1].iov_base = buff;
(12)     iov[1].iov_len = bufflen;
(13)     eio.iov = iov;
(14)     eio.binv = binv;
(15)     eio.vsize = 2;
(16)     eio.size = bufflen + 4;
(17)     written = 0;
(18)     if (driver_sizeq(ud->port) == 0) {
(19)         if ((written = writev(ud->fd, iov, 2)) == eio.size) {
(20)             ud->sent += written;
(21)             if (ud->type == portTypeCommand) {
(22)                 driver_output(ud->port, "Sok", 3);
(23)             }
(24)             return;
(25)         } else if (written < 0) {
(26)             if (errno != EWOULDBLOCK) {
(27)                 driver_failure_eof(ud->port);
(28)                 return;
(29)             } else {
(30)                 written = 0;
(31)             }
(32)         } else {
```

```
(33)             ud->sent += written;
(34)         }
(35)         /* Enqueue remaining */
(36)     }
(37)     driver_enqv(ud->port, &eio, written);
(38)     send_out_queue(ud);
(39) }
```

This driver uses the `writev` system call to send data onto the socket. A combination of writev and the driver output queues is very convenient. An *ErlIOVec* structure contains a *SysIOVec* (which is equivalent to the `struct iovec` structure defined in `uio.h`. The ErlIOVec also contains an array of *ErlDrvBinary* pointers, of the same length as the number of buffers in the I/O vector itself. One can use this to allocate the binaries for the queue "manually" in the driver, but we'll just fill the binary array with NULL values (line 7) , which will make the runtime system allocate it's own buffers when we call `driver_enqv` (line 37).

The routine builds an I/O vector containing the header bytes and the buffer (the opcode has been removed and the buffer length decreased by the output routine). If the queue is empty, we'll write the data directly to the socket (or at least try to). If any data is left, it is stored in the queue and then we try to send the queue (line 38). An ack is sent when the message is delivered completely (line 22). The `send_out_queue` will send acks if the sending is completed there. If the port is in command mode, the Erlang code serializes the send operations so that only one packet can be waiting for delivery at a time. Therefore the ack can be sent simply whenever the queue is empty.

A short look at the `send_out_queue` routine:

```
( 1) static int send_out_queue(UdsData *ud)
( 2) {
( 3)     for(;;) {
( 4)         int vlen;
( 5)         SysIOVec *tmp = driver_peekq(ud->port, &vlen);
( 6)         int wrote;
( 7)         if (tmp == NULL) {
( 8)             driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_WRITE, 0);
( 9)             if (ud->type == portTypeCommand) {
(10)                 driver_output(ud->port, "Sok", 3);
(11)             }
(12)             return 0;
(13)         }
(14)         if (vlen > IO_VECTOR_MAX) {
(15)             vlen = IO_VECTOR_MAX;
(16)         }
(17)         if ((wrote = writev(ud->fd, tmp, vlen)) < 0) {
(18)             if (errno == EWOULDBLOCK) {
(19)                 driver_select(ud->port, (ErlDrvEvent) ud->fd,
(20)                               DO_WRITE, 1);
(21)                 return 0;
(22)             } else {
(23)                 driver_failure_eof(ud->port);
(24)                 return -1;
(25)             }
(26)         }
(27)         driver_deq(ud->port, wrote);
(28)         ud->sent += wrote;
(29)     }
(30) }
```

What we do is simply to pick out an I/O vector from the queue (which is the whole queue as an *SysIOVec*). If the I/O vector is to long (IO_VECTOR_MAX is defined to 16), the vector length is decreased (line 15), otherwise the `writev` (line 17) call will fail. Writing is tried and anything written is dequeued (line 27). If the write fails with `EWOULDBLOCK`

(note that all sockets are in nonblocking mode), `driver_select` is called to make the `uds_output` routine be called when there is space to write again.

We will continue trying to write until the queue is empty or the writing would block.

The routine above are called from the `uds_output` routine, which looks like this:

```
( 1) static void uds_output(ErlDrvData handle, ErlDrvEvent event)
( 2) {
( 3)    UdsData *ud = (UdsData *) handle;
( 4)    if (ud->type == portTypeConnector) {
( 5)        ud->type = portTypeCommand;
( 6)        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_WRITE, 0);
( 7)        driver_output(ud->port, "Cok",3);
( 8)        return;
( 9)    }
(10)    send_out_queue(ud);
(11) }
```

The routine is simple, it first handles the fact that the output select will concern a socket in the business of connecting (and the connecting blocked). If the socket is in a connected state it simply sends the output queue, this routine is called when there is possible to write to a socket where we have an output queue, so there is no question what to do.

The driver implements a control interface, which is a synchronous interface called when Erlang calls `erlang:port_control/3`. This is the only interface that can control the driver when it is in data mode and it may be called with the following opcodes:

- 'C': Set port in command mode.
- 'I': Set port in intermediate mode.
- 'D': Set port in data mode.
- 'N': Get identification number for listen port, this identification number is used in an accept command to the driver, it is returned as a big endian 32 bit integer, which happens to be the file identifier for the listen socket.
- 'S': Get statistics, which is the number of bytes received, the number of bytes sent and the number of bytes pending in the output queue. This data is used when the distribution checks that a connection is alive (ticking). The statistics is returned as 3 32 bit big endian integers.
- 'T': Send a tick message, which is a packet of length 0. Ticking is done when the port is in data mode, so the command for sending data cannot be used (besides it ignores zero length packages in command mode). This is used by the ticker to send dummy data when no other traffic is present. *Note* that it is important that the interface for sending ticks is not blocking. This implementation uses `erlang:port_control/3` which does not block the caller. If `erlang:port_command` is used, use `erlang:port_command/3` and pass `[force]` as option list; otherwise, the caller can be blocked indefinitely on a busy port and prevent the system from taking down a connection that is not functioning.
- 'R': Get creation number of listen socket, which is used to dig out the number stored in the lock file to differentiate between invocations of Erlang nodes with the same name.

The control interface gets a buffer to return its value in, but is free to allocate it's own buffer is the provided one is to small. Here is the code for `uds_control`:

```
( 1) static int uds_control(ErlDrvData handle, unsigned int command,
( 2)                        char* buf, int count, char** res, int res_size)
( 3) {
( 4) /* Local macro to ensure large enough buffer. */
( 5) #define ENSURE(N)                                \
( 6)    do {                                         \
( 7)        if (res_size < N) {                      \
```

```
( 8)            *res = ALLOC(N);                    \
( 9)        }                                       \
(10)    } while(0)

(11)    UdsData *ud = (UdsData *) handle;

(12)    switch (command) {
(13)    case 'S':
(14)        {
(15)            ENSURE(13);
(16)            **res = 0;
(17)            put_packet_length((*res) + 1, ud->received);
(18)            put_packet_length((*res) + 5, ud->sent);
(19)            put_packet_length((*res) + 9, driver_sizeq(ud->port));
(20)            return 13;
(21)        }
(22)    case 'C':
(23)        if (ud->type < portTypeCommand) {
(24)            return report_control_error(res, res_size, "einval");
(25)        }
(26)        ud->type = portTypeCommand;
(27)        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
(28)        ENSURE(1);
(29)        **res = 0;
(30)        return 1;
(31)    case 'I':
(32)        if (ud->type < portTypeCommand) {
(33)            return report_control_error(res, res_size, "einval");
(34)        }
(35)        ud->type = portTypeIntermediate;
(36)        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
(37)        ENSURE(1);
(38)        **res = 0;
(39)        return 1;
(40)    case 'D':
(41)        if (ud->type < portTypeCommand) {
(42)            return report_control_error(res, res_size, "einval");
(43)        }
(44)        ud->type = portTypeData;
(45)        do_recv(ud);
(46)        ENSURE(1);
(47)        **res = 0;
(48)        return 1;
(49)    case 'N':
(50)        if (ud->type != portTypeListener) {
(51)            return report_control_error(res, res_size, "einval");
(52)        }
(53)        ENSURE(5);
(54)        (*res)[0] = 0;
(55)        put_packet_length((*res) + 1, ud->fd);
(56)        return 5;
(57)    case 'T': /* tick */
(58)        if (ud->type != portTypeData) {
(59)            return report_control_error(res, res_size, "einval");
(60)        }
(61)        do_send(ud,"",0);
(62)        ENSURE(1);
(63)        **res = 0;
(64)        return 1;
(65)    case 'R':
(66)        if (ud->type != portTypeListener) {
(67)            return report_control_error(res, res_size, "einval");
(68)        }
(69)        ENSURE(2);
(70)        (*res)[0] = 0;
```

```
(71)        (*res)[1] = ud->creation;
(72)        return 2;
(73)    default:
(74)        return report_control_error(res, res_size, "einval");
(75)    }
(76) #undef ENSURE
(77) }
```

The macro ENSURE (line 5 to 10) is used to ensure that the buffer is large enough for our answer. We switch on the command and take actions, there is not much to say about this routine. Worth noting is that we always has read select active on a port in data mode (achieved by calling `do_recv` on line 45), but turn off read selection in intermediate and command modes (line 27 and 36).

The rest of the driver is more or less UDS specific and not of general interest.

1.3.3 Putting it all together

To test the distribution, one can use the `net_kernel:start/1` function, which is useful as it starts the distribution on a running system, where tracing/debugging can be performed. The `net_kernel:start/1` routine takes a list as it's single argument. The lists first element should be the node name (without the "@hostname") as an atom, and the second (and last) element should be one of the atoms `shortnames` or `longnames`. In the example case `shortnames` is preferred.

For net kernel to find out which distribution module to use, the command line argument `-proto_dist` is used. The argument is followed by one or more distribution module names, with the "_dist" suffix removed, i.e. uds_dist as a distribution module is specified as `-proto_dist uds`.

If no epmd (TCP port mapper daemon) is used, one should also specify the command line option `-no_epmd`, which will make Erlang skip the epmd startup, both as a OS process and as an Erlang ditto.

The path to the directory where the distribution modules reside must be known at boot, which can either be achieved by specifying `-pa <path>` on the command line or by building a boot script containing the applications used for your distribution protocol (in the uds_dist protocol, it's only the uds_dist application that needs to be added to the script).

The distribution will be started at boot if all the above is specified and an `-sname <name>` flag is present at the command line, here follows two examples:

```
$ erl -pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin -proto_dist uds -no_epmd
Erlang (BEAM) emulator version 5.0

Eshell V5.0  (abort with ^G)
1> net_kernel:start([bing,shortnames]).
{ok,<0.30.0>}
(bing@hador)2>
```

...

```
$ erl -pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin -proto_dist uds \
      -no_epmd -sname bong
Erlang (BEAM) emulator version 5.0

Eshell V5.0  (abort with ^G)
(bong@hador)1>
```

One can utilize the ERL_FLAGS environment variable to store the complicated parameters in:

```
$ ERL_FLAGS=-pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin \
      -proto_dist uds -no_epmd
$ export ERL_FLAGS
$ erl -sname bang
Erlang (BEAM) emulator version 5.0

Eshell V5.0  (abort with ^G)
(bang@hador)1>
```

The ERL_FLAGS should preferably not include the name of the node.

1.4 The Abstract Format

This document describes the standard representation of parse trees for Erlang programs as Erlang terms. This representation is known as the *abstract format*. Functions dealing with such parse trees are `compile:forms/[1,2]` and functions in the modules `epp`, `erl_eval`, `erl_lint`, `erl_pp`, `erl_parse`, and `io`. They are also used as input and output for parse transforms (see the module `compile`).

We use the function `Rep` to denote the mapping from an Erlang source construct `C` to its abstract format representation `R`, and write `R = Rep(C)`.

The word `LINE` below represents an integer, and denotes the number of the line in the source file where the construction occurred. Several instances of `LINE` in the same construction may denote different lines.

Since operators are not terms in their own right, when operators are mentioned below, the representation of an operator should be taken to be the atom with a printname consisting of the same characters as the operator.

1.4.1 Module declarations and forms

A module declaration consists of a sequence of forms that are either function declarations or attributes.

- If D is a module declaration consisting of the forms `F_1`, ..., `F_k`, then Rep(D) = `[Rep(F_1), ..., Rep(F_k)]`.
- If F is an attribute `-module(Mod)`, then Rep(F) = `{attribute,LINE,module,Mod}`.
- If F is an attribute `-export([Fun_1/A_1, ..., Fun_k/A_k])`, then Rep(F) = `{attribute,LINE,export,[{Fun_1,A_1}, ..., {Fun_k,A_k}]}`.
- If F is an attribute `-import(Mod,[Fun_1/A_1, ..., Fun_k/A_k])`, then Rep(F) = `{attribute,LINE,import,{Mod,[{Fun_1,A_1}, ..., {Fun_k,A_k}]}}`.
- If F is an attribute `-compile(Options)`, then Rep(F) = `{attribute,LINE,compile,Options}`.
- If F is an attribute `-file(File,Line)`, then Rep(F) = `{attribute,LINE,file,{File,Line}}`.
- If F is a record declaration `-record(Name,{V_1, ..., V_k})`, then Rep(F) = `{attribute,LINE,record,{Name,[Rep(V_1), ..., Rep(V_k)]}}`. For Rep(V), see below.
- If F is a wild attribute `-A(T)`, then Rep(F) = `{attribute,LINE,A,T}`.
- If F is a function declaration `Name Fc_1 ; ... ; Name Fc_k`, where each `Fc_i` is a function clause with a pattern sequence of the same length `Arity`, then Rep(F) = `{function,LINE,Name,Arity, [Rep(Fc_1), ...,Rep(Fc_k)]}`.

Record fields

Each field in a record declaration may have an optional explicit default initializer expression

- If V is `A`, then Rep(V) = `{record_field,LINE,Rep(A)}`.
- If V is `A = E`, then Rep(V) = `{record_field,LINE,Rep(A),Rep(E)}`.

Representation of parse errors and end of file

In addition to the representations of forms, the list that represents a module declaration (as returned by functions in `erl_parse` and `epp`) may contain tuples `{error,E}` and `{warning,W}`, denoting syntactically incorrect forms and warnings, and `{eof,LINE}`, denoting an end of stream encountered before a complete form had been parsed.

1.4.2 Atomic literals

There are five kinds of atomic literals, which are represented in the same way in patterns, expressions and guards:

- If L is an integer or character literal, then Rep(L) = `{integer,LINE,L}`.
- If L is a float literal, then Rep(L) = `{float,LINE,L}`.
- If L is a string literal consisting of the characters `C_1`, ..., `C_k`, then Rep(L) = `{string,LINE, [C_1, ..., C_k]}`.
- If L is an atom literal, then Rep(L) = `{atom,LINE,L}`.

Note that negative integer and float literals do not occur as such; they are parsed as an application of the unary negation operator.

1.4.3 Patterns

If `Ps` is a sequence of patterns `P_1, ..., P_k`, then Rep(Ps) = `[Rep(P_1), ..., Rep(P_k)]`. Such sequences occur as the list of arguments to a function or fun.

Individual patterns are represented as follows:

- If P is an atomic literal L, then Rep(P) = Rep(L).
- If P is a compound pattern `P_1 = P_2`, then Rep(P) = `{match,LINE,Rep(P_1),Rep(P_2)}`.
- If P is a variable pattern `V`, then Rep(P) = `{var,LINE,A}`, where A is an atom with a printname consisting of the same characters as `V`.
- If P is a universal pattern _, then Rep(P) = `{var,LINE,'_'}`.
- If P is a tuple pattern `{P_1, ..., P_k}`, then Rep(P) = `{tuple,LINE,[Rep(P_1), ..., Rep(P_k)]}`.
- If P is a nil pattern `[]`, then Rep(P) = `{nil,LINE}`.
- If P is a cons pattern `[P_h | P_t]`, then Rep(P) = `{cons,LINE,Rep(P_h),Rep(P_t)}`.
- If E is a binary pattern `<<P_1:Size_1/TSL_1, ..., P_k:Size_k/TSL_k>>`, then Rep(E) = `{bin,LINE,[{bin_element,LINE,Rep(P_1),Rep(Size_1),Rep(TSL_1)}, ..., {bin_element,LINE,Rep(P_k),Rep(Size_k),Rep(TSL_k)}]}`. For Rep(TSL), see below. An omitted `Size` is represented by `default`. An omitted `TSL` (type specifier list) is represented by `default`.
- If P is `P_1 Op P_2`, where `Op` is a binary operator (this is either an occurrence of `++` applied to a literal string or character list, or an occurrence of an expression that can be evaluated to a number at compile time), then Rep(P) = `{op,LINE,Op,Rep(P_1),Rep(P_2)}`.
- If P is `Op P_0`, where `Op` is a unary operator (this is an occurrence of an expression that can be evaluated to a number at compile time), then Rep(P) = `{op,LINE,Op,Rep(P_0)}`.
- If P is a record pattern `#Name{Field_1=P_1, ..., Field_k=P_k}`, then Rep(P) = `{record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(P_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(P_k)}]}`.
- If P is `#Name.Field`, then Rep(P) = `{record_index,LINE,Name,Rep(Field)}`.
- If P is `( P_0 )`, then Rep(P) = `Rep(P_0)`, i.e., patterns cannot be distinguished from their bodies.

Note that every pattern has the same source form as some expression, and is represented the same way as the corresponding expression.

1.4.4 Expressions

A body B is a sequence of expressions `E_1, ..., E_k`, and Rep(B) = `[Rep(E_1), ..., Rep(E_k)]`.

An expression E is one of the following alternatives:

- If P is an atomic literal `L`, then Rep(P) = Rep(L).
- If E is `P = E_0`, then Rep(E) = `{match,LINE,Rep(P),Rep(E_0)}`.
- If E is a variable `V`, then Rep(E) = `{var,LINE,A}`, where `A` is an atom with a printname consisting of the same characters as `V`.
- If E is a tuple skeleton `{E_1, ..., E_k}`, then Rep(E) = `{tuple,LINE,[Rep(E_1), ..., Rep(E_k)]}`.
- If E is `[]`, then Rep(E) = `{nil,LINE}`.
- If E is a cons skeleton `[E_h | E_t]`, then Rep(E) = `{cons,LINE,Rep(E_h),Rep(E_t)}`.
- If E is a binary constructor `<<V_1:Size_1/TSL_1, ..., V_k:Size_k/TSL_k>>`, then Rep(E) = `{bin,LINE,[{bin_element,LINE,Rep(V_1),Rep(Size_1),Rep(TSL_1)}, ..., {bin_element,LINE,Rep(V_k),Rep(Size_k),Rep(TSL_k)}]}`. For Rep(TSL), see below. An omitted `Size` is represented by `default`. An omitted `TSL` (type specifier list) is represented by `default`.
- If E is `E_1 Op E_2`, where `Op` is a binary operator, then Rep(E) = `{op,LINE,Op,Rep(E_1),Rep(E_2)}`.
- If E is `Op E_0`, where `Op` is a unary operator, then Rep(E) = `{op,LINE,Op,Rep(E_0)}`.
- If E is `#Name{Field_1=E_1, ..., Field_k=E_k}`, then Rep(E) = `{record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(E_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(E_k)}]}`.
- If E is `E_0#Name{Field_1=E_1, ..., Field_k=E_k}`, then Rep(E) = `{record,LINE,Rep(E_0),Name, [{record_field,LINE,Rep(Field_1),Rep(E_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(E_k)}]}`.
- If E is `#Name.Field`, then Rep(E) = `{record_index,LINE,Name,Rep(Field)}`.
- If E is `E_0#Name.Field`, then Rep(E) = `{record_field,LINE,Rep(E_0),Name,Rep(Field)}`.
- If E is `catch E_0`, then Rep(E) = `{'catch',LINE,Rep(E_0)}`.
- If E is `E_0(E_1, ..., E_k)`, then Rep(E) = `{call,LINE,Rep(E_0),[Rep(E_1), ..., Rep(E_k)]}`.
- If E is `E_m:E_0(E_1, ..., E_k)`, then Rep(E) = `{call,LINE, {remote,LINE,Rep(E_m),Rep(E_0)},[Rep(E_1), ..., Rep(E_k)]}`.
- If E is a list comprehension `[E_0 || W_1, ..., W_k]`, where each `W_i` is a generator or a filter, then Rep(E) = `{lc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}`. For Rep(W), see below.
- If E is a binary comprehension `<<E_0 || W_1, ..., W_k>>`, where each `W_i` is a generator or a filter, then Rep(E) = `{bc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}`. For Rep(W), see below.
- If E is `begin B end`, where `B` is a body, then Rep(E) = `{block,LINE,Rep(B)}`.
- If E is `if Ic_1 ; ... ; Ic_k end`, where each `Ic_i` is an if clause then Rep(E) = `{'if',LINE, [Rep(Ic_1), ..., Rep(Ic_k)]}`.
- If E is `case E_0 of Cc_1 ; ... ; Cc_k end`, where `E_0` is an expression and each `Cc_i` is a case clause then Rep(E) = `{'case',LINE,Rep(E_0),[Rep(Cc_1), ..., Rep(Cc_k)]}`.
- If E is `try B catch Tc_1 ; ... ; Tc_k end`, where `B` is a body and each `Tc_i` is a catch clause then Rep(E) = `{'try',LINE,Rep(B),[],[Rep(Tc_1), ..., Rep(Tc_k)],[]}`.
- If E is `try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n end`, where `B` is a body, each `Cc_i` is a case clause and each `Tc_j` is a catch clause then Rep(E) = `{'try',LINE,Rep(B), [Rep(Cc_1), ..., Rep(Cc_k)],[Rep(Tc_1), ..., Rep(Tc_n)],[]}`.

- If E is `try B after A end`, where `B` and `A` are bodies then Rep(E) = `{'try',LINE,Rep(B),[],[],Rep(A)}`.
- If E is `try B of Cc_1 ; ... ; Cc_k after A end`, where `B` and `A` are a bodies and each `Cc_i` is a case clause then Rep(E) = `{'try',LINE,Rep(B),[Rep(Cc_1), ..., Rep(Cc_k)],[],Rep(A)}`.
- If E is `try B catch Tc_1 ; ... ; Tc_k after A end`, where `B` and `A` are bodies and each `Tc_i` is a catch clause then Rep(E) = `{'try',LINE,Rep(B),[],[Rep(Tc_1), ..., Rep(Tc_k)],Rep(A)}`.
- If E is `try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n after A end`, where `B` and `A` are a bodies, each `Cc_i` is a case clause and each `Tc_j` is a catch clause then Rep(E) = `{'try',LINE,Rep(B),[Rep(Cc_1), ..., Rep(Cc_k)],[Rep(Tc_1), ..., Rep(Tc_n)],Rep(A)}`.
- If E is `receive Cc_1 ; ... ; Cc_k end`, where each `Cc_i` is a case clause then Rep(E) = `{'receive',LINE,[Rep(Cc_1), ..., Rep(Cc_k)]}`.
- If E is `receive Cc_1 ; ... ; Cc_k after E_0 -> B_t end`, where each `Cc_i` is a case clause, `E_0` is an expression and `B_t` is a body, then Rep(E) = `{'receive',LINE,[Rep(Cc_1), ..., Rep(Cc_k)],Rep(E_0),Rep(B_t)}`.
- If E is `fun Name / Arity`, then Rep(E) = `{'fun',LINE,{function,Name,Arity}}`.
- If E is `fun Module:Name/Arity`, then Rep(E) = `{'fun',LINE,{function,Module,Name,Arity}}`.
- If E is `fun Fc_1 ; ... ; Fc_k end` where each `Fc_i` is a function clause then Rep(E) = `{'fun',LINE,{clauses,[Rep(Fc_1), ..., Rep(Fc_k)]}}`.
- If E is `query [E_0 || W_1, ..., W_k] end`, where each `W_i` is a generator or a filter, then Rep(E) = `{'query',LINE,{lc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}}`. For Rep(W), see below.
- If E is `E_0.Field`, a Mnesia record access inside a query, then Rep(E) = `{record_field,LINE,Rep(E_0),Rep(Field)}`.
- If E is `( E_0 )`, then Rep(E) = `Rep(E_0)`, i.e., parenthesized expressions cannot be distinguished from their bodies.

Generators and filters

When W is a generator or a filter (in the body of a list or binary comprehension), then:

- If W is a generator `P <- E`, where `P` is a pattern and `E` is an expression, then Rep(W) = `{generate,LINE,Rep(P),Rep(E)}`.
- If W is a generator `P <= E`, where `P` is a pattern and `E` is an expression, then Rep(W) = `{b_generate,LINE,Rep(P),Rep(E)}`.
- If W is a filter `E`, which is an expression, then Rep(W) = `Rep(E)`.

Binary element type specifiers

A type specifier list TSL for a binary element is a sequence of type specifiers `TS_1 - ... - TS_k`. Rep(TSL) = `[Rep(TS_1), ..., Rep(TS_k)]`.

When TS is a type specifier for a binary element, then:

- If TS is an atom `A`, Rep(TS) = `A`.
- If TS is a couple `A:Value` where `A` is an atom and `Value` is an integer, Rep(TS) = `{A, Value}`.

1.4.5 Clauses

There are function clauses, if clauses, case clauses and catch clauses.

A clause C is one of the following alternatives:

- If C is a function clause `( Ps ) -> B` where Ps is a pattern sequence and B is a body, then Rep(C) = `{clause,LINE,Rep(Ps),[],Rep(B)}`.
- If C is a function clause `( Ps ) when Gs -> B` where Ps is a pattern sequence, Gs is a guard sequence and B is a body, then Rep(C) = `{clause,LINE,Rep(Ps),Rep(Gs),Rep(B)}`.
- If C is an if clause `Gs -> B` where Gs is a guard sequence and B is a body, then Rep(C) = `{clause,LINE,[],Rep(Gs),Rep(B)}`.
- If C is a case clause `P -> B` where P is a pattern and B is a body, then Rep(C) = `{clause,LINE,[Rep(P)],[],Rep(B)}`.
- If C is a case clause `P when Gs -> B` where P is a pattern, Gs is a guard sequence and B is a body, then Rep(C) = `{clause,LINE,[Rep(P)],Rep(Gs),Rep(B)}`.
- If C is a catch clause `P -> B` where P is a pattern and B is a body, then Rep(C) = `{clause,LINE,[Rep({throw,P,_})],[],Rep(B)}`.
- If C is a catch clause `X : P -> B` where X is an atomic literal or a variable pattern, P is a pattern and B is a body, then Rep(C) = `{clause,LINE,[Rep({X,P,_})],[],Rep(B)}`.
- If C is a catch clause `P when Gs -> B` where P is a pattern, Gs is a guard sequence and B is a body, then Rep(C) = `{clause,LINE,[Rep({throw,P,_})],Rep(Gs),Rep(B)}`.
- If C is a catch clause `X : P when Gs -> B` where X is an atomic literal or a variable pattern, P is a pattern, Gs is a guard sequence and B is a body, then Rep(C) = `{clause,LINE,[Rep({X,P,_})],Rep(Gs),Rep(B)}`.

1.4.6 Guards

A guard sequence Gs is a sequence of guards `G_1; ...; G_k`, and Rep(Gs) = `[Rep(G_1), ..., Rep(G_k)]`. If the guard sequence is empty, Rep(Gs) = `[]`.

A guard G is a nonempty sequence of guard tests `Gt_1, ..., Gt_k`, and Rep(G) = `[Rep(Gt_1), ..., Rep(Gt_k)]`.

A guard test Gt is one of the following alternatives:

- If Gt is an atomic literal L, then Rep(Gt) = Rep(L).
- If Gt is a variable pattern V, then Rep(Gt) = `{var,LINE,A}`, where A is an atom with a printname consisting of the same characters as V.
- If Gt is a tuple skeleton `{Gt_1, ..., Gt_k}`, then Rep(Gt) = `{tuple,LINE,[Rep(Gt_1), ..., Rep(Gt_k)]}`.
- If Gt is `[]`, then Rep(Gt) = `{nil,LINE}`.
- If Gt is a cons skeleton `[Gt_h | Gt_t]`, then Rep(Gt) = `{cons,LINE,Rep(Gt_h),Rep(Gt_t)}`.
- If Gt is a binary constructor `<<Gt_1:Size_1/TSL_1, ..., Gt_k:Size_k/TSL_k>>`, then Rep(Gt) = `{bin,LINE,[{bin_element,LINE,Rep(Gt_1),Rep(Size_1),Rep(TSL_1)}, ..., {bin_element,LINE,Rep(Gt_k),Rep(Size_k),Rep(TSL_k)}]}`. For Rep(TSL), see above. An omitted `Size` is represented by `default`. An omitted `TSL` (type specifier list) is represented by `default`.
- If Gt is `Gt_1 Op Gt_2`, where Op is a binary operator, then Rep(Gt) = `{op,LINE,Op,Rep(Gt_1),Rep(Gt_2)}`.
- If Gt is `Op Gt_0`, where Op is a unary operator, then Rep(Gt) = `{op,LINE,Op,Rep(Gt_0)}`.
- If Gt is `#Name{Field_1=Gt_1, ..., Field_k=Gt_k}`, then Rep(E) = `{record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(Gt_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(Gt_k)}]}`.
- If Gt is `#Name.Field`, then Rep(Gt) = `{record_index,LINE,Name,Rep(Field)}`.

- If Gt is `Gt_0#Name.Field`, then Rep(Gt) = `{record_field,LINE,Rep(Gt_0),Name,Rep(Field)}`.
- If Gt is `A(Gt_1, ..., Gt_k)`, where `A` is an atom, then Rep(Gt) = `{call,LINE,Rep(A),[Rep(Gt_1), ..., Rep(Gt_k)]}`.
- If Gt is `A_m:A(Gt_1, ..., Gt_k)`, where `A_m` is the atom `erlang` and `A` is an atom or an operator, then Rep(Gt) = `{call,LINE,{remote,LINE,Rep(A_m),Rep(A)},[Rep(Gt_1), ..., Rep(Gt_k)]}`.
- If Gt is `{A_m,A}(Gt_1, ..., Gt_k)`, where `A_m` is the atom `erlang` and `A` is an atom or an operator, then Rep(Gt) = `{call,LINE,Rep({A_m,A}),[Rep(Gt_1), ..., Rep(Gt_k)]}`.
- If Gt is `( Gt_0 )`, then Rep(Gt) = `Rep(Gt_0)`, i.e., parenthesized guard tests cannot be distinguished from their bodies.

Note that every guard test has the same source form as some expression, and is represented the same way as the corresponding expression.

1.4.7 The abstract format after preprocessing

The compilation option `debug_info` can be given to the compiler to have the abstract code stored in the `abstract_code` chunk in the BEAM file (for debugging purposes).

In OTP R9C and later, the `abstract_code` chunk will contain

`{raw_abstract_v1,AbstractCode}`

where `AbstractCode` is the abstract code as described in this document.

In releases of OTP prior to R9C, the abstract code after some more processing was stored in the BEAM file. The first element of the tuple would be either `abstract_v1` (R7B) or `abstract_v2` (R8B).

1.5 tty - A command line interface

`tty` is a simple command line interface program where keystrokes are collected and interpreted. Completed lines are sent to the shell for interpretation. There is a simple history mechanism, which saves previous lines. These can be edited before sending them to the shell. `tty` is started when Erlang is started with the command:

erl

`tty` operates in one of two modes:

- *normal mode*, in which lines of text can be edited and sent to the shell.
- *shell break* mode, which allows the user to kill the current shell, start multiple shells etc. Shell break mode is started by typing *Control G*.

1.5.1 Normal Mode

In normal mode keystrokes from the user are collected and interpreted by `tty`. Most of the *emacs* line editing commands are supported. The following is a complete list of the supported line editing commands.

Note: The notation `C-a` means pressing the control key and the letter `a` simultaneously. `M-f` means pressing the `ESC` key followed by the letter `f`.

Key Sequence	*Function*
C-a	Beginning of line
C-b	Backward character

M-b	Backward word
C-d	Delete character
M-d	Delete word
C-e	End of line
C-f	Forward character
M-f	Forward word
C-g	Enter shell break mode
C-k	Kill line
C-l	Redraw line
C-n	Fetch next line from the history buffer
C-p	Fetch previous line from the history buffer
C-t	Transpose characters
C-y	Insert previously killed text

Table 5.1: tty text editing

1.5.2 Shell Break Mode

tty enters *shell* break mode when you type *Control G*. In this mode you can:

- Kill or suspend the current shell
- Connect to a suspended shell
- Start a new shell

1.6 How to implement a driver

> Note:
>
> This document was written a long time ago. A lot of it is still valid, but some things have changed since it was first written. Updates of this document are planned for the future. The reader is encouraged to also read the *erl_driver*, and the *driver_entry* documentation.

1.6.1 Introduction

This chapter tells you how to build your own driver for erlang.

A driver in Erlang is a library written in C, that is linked to the Erlang emulator and called from erlang. Drivers can be used when C is more suitable than Erlang, to speed things up, or to provide access to OS resources not directly accessible from Erlang.

A driver can be dynamically loaded, as a shared library (known as a DLL on windows), or statically loaded, linked with the emulator when it is compiled and linked. Only dynamically loaded drivers are described here, statically linked drivers are beyond the scope of this chapter.

When a driver is loaded it is executed in the context of the emulator, shares the same memory and the same thread. This means that all operations in the driver must be non-blocking, and that any crash in the driver will bring the whole emulator down. In short: you have to be extremely careful!

1.6.2 Sample driver

This is a simple driver for accessing a postgres database using the libpq C client library. Postgres is used because it's free and open source. For information on postgres, refer to the website www.postgres.org.

The driver is synchronous, it uses the synchronous calls of the client library. This is only for simplicity, and is generally not good, since it will halt the emulator while waiting for the database. This will be improved on below with an asynchronous sample driver.

The code is quite straight-forward: all communication between Erlang and the driver is done with `port_control/3`, and the driver returns data back using the `rbuf`.

An Erlang driver only exports one function: the driver entry function. This is defined with a macro, `DRIVER_INIT`, and returns a pointer to a C `struct` containing the entry points that are called from the emulator. The `struct` defines the entries that the emulator calls to call the driver, with a `NULL` pointer for entries that are not defined and used by the driver.

The `start` entry is called when the driver is opened as a port with `open_port/2`. Here we allocate memory for a user data structure. This user data will be passed every time the emulator calls us. First we store the driver handle, because it is needed in subsequent calls. We allocate memory for the connection handle that is used by LibPQ. We also set the port to return allocated driver binaries, by setting the flag `PORT_CONTROL_FLAG_BINARY`, calling `set_port_control_flags`. (This is because we don't know whether our data will fit in the result buffer of `control`, which has a default size set up by the emulator, currently 64 bytes.)

There is an entry `init` which is called when the driver is loaded, but we don't use this, since it is executed only once, and we want to have the possibility of several instances of the driver.

The `stop` entry is called when the port is closed.

The `control` entry is called from the emulator when the Erlang code calls `port_control/3`, to do the actual work. We have defined a simple set of commands: `connect` to login to the database, `disconnect` to log out and `select` to send a SQL-query and get the result. All results are returned through `rbuf`. The library `ei` in `erl_interface` is used to encode data in binary term format. The result is returned to the emulator as binary terms, so `binary_to_term` is called in Erlang to convert the result to term form.

The code is available in `pg_sync.c` in the `sample` directory of `erts`.

The driver entry contains the functions that will be called by the emulator. In our simple example, we only provide `start`, `stop` and `control`.

```
/* Driver interface declarations */
static ErlDrvData start(ErlDrvPort port, char *command);
static void stop(ErlDrvData drv_data);
static int control(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);

static ErlDrvEntry pq_driver_entry = {
```

```
    NULL,                       /* init */
    start,
    stop,
    NULL,                       /* output */
    NULL,                       /* ready_input */
    NULL,                       /* ready_output */
    "pg_sync",                  /* the name of the driver */
    NULL,                       /* finish */
    NULL,                       /* handle */
    control,
    NULL,                       /* timeout */
    NULL,                       /* outputv */
    NULL,                       /* ready_async */
    NULL,                       /* flush */
    NULL,                       /* call */
    NULL                        /* event */
};
```

We have a structure to store state needed by the driver, in this case we only need to keep the database connection.

```
typedef struct our_data_s {
    PGconn* conn;
} our_data_t;
```

These are control codes we have defined.

```
/* Keep the following definitions in alignment with the
 * defines in erl_pq_sync.erl
 */

#define DRV_CONNECT             'C'
#define DRV_DISCONNECT          'D'
#define DRV_SELECT              'S'
```

This just returns the driver structure. The macro `DRIVER_INIT` defines the only exported function. All the other functions are static, and will not be exported from the library.

```
/* INITIALIZATION AFTER LOADING */

/*
 * This is the init function called after this driver has been loaded.
 * It must *not* be declared static. Must return the address to
 * the driver entry.
 */

DRIVER_INIT(pq_drv)
{
    return &pq_driver_entry;
}
```

Here we do some initialization, `start` is called from `open_port`. The data will be passed to `control` and `stop`.

```
/* DRIVER INTERFACE */
static ErlDrvData start(ErlDrvPort port, char *command)
{
    our_data_t* data;

    data = (our_data_t*)driver_alloc(sizeof(our_data_t));
    data->conn = NULL;
    set_port_control_flags(port, PORT_CONTROL_FLAG_BINARY);
    return (ErlDrvData)data;
}
```

We call disconnect to log out from the database. (This should have been done from Erlang, but just in case.)

```
 static int do_disconnect(our_data_t* data, ei_x_buff* x);

static void stop(ErlDrvData drv_data)
{
    do_disconnect((our_data_t*)drv_data, NULL);
}
```

We use the binary format only to return data to the emulator; input data is a string paramater for `connect` and `select`. The returned data consists of Erlang terms.

The functions `get_s` and `ei_x_to_new_binary` are utilities that is used to make the code shorter. `get_s` duplicates the string and zero-terminates it, since the postgres client library wants that. `ei_x_to_new_binary` takes an `ei_x_buff` buffer and allocates a binary and copies the data there. This binary is returned in `*rbuf`. (Note that this binary is freed by the emulator, not by us.)

```
static char* get_s(const char* buf, int len);
static int do_connect(const char *s, our_data_t* data, ei_x_buff* x);
static int do_select(const char* s, our_data_t* data, ei_x_buff* x);

/* Since we are operating in binary mode, the return value from control
 * is irrelevant, as long as it is not negative.
 */
static int control(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen)
{
    int r;
    ei_x_buff x;
    our_data_t* data = (our_data_t*)drv_data;
    char* s = get_s(buf, len);
    ei_x_new_with_version(&x);
    switch (command) {
        case DRV_CONNECT:    r = do_connect(s, data, &x);  break;
        case DRV_DISCONNECT: r = do_disconnect(data, &x);  break;
        case DRV_SELECT:     r = do_select(s, data, &x);   break;
        default:             r = -1;        break;
    }
    *rbuf = (char*)ei_x_to_new_binary(&x);
    ei_x_free(&x);
    driver_free(s);
    return r;
}
```

In `do_connect` is where we log in to the database. If the connection was successful we store the connection handle in our driver data, and return ok. Otherwise, we return the error message from postgres, and store `NULL` in the driver data.

```
static int do_connect(const char *s, our_data_t* data, ei_x_buff* x)
{
    PGconn* conn = PQconnectdb(s);
    if (PQstatus(conn) != CONNECTION_OK) {
        encode_error(x, conn);
        PQfinish(conn);
        conn = NULL;
    } else {
        encode_ok(x);
    }
    data->conn = conn;
    return 0;
}
```

If we are connected (if the connection handle is not `NULL`), we log out from the database. We need to check if a we should encode an ok, since we might get here from the `stop` function, which doesn't return data to the emulator.

```
static int do_disconnect(our_data_t* data, ei_x_buff* x)
{
    if (data->conn == NULL)
        return 0;
    PQfinish(data->conn);
    data->conn = NULL;
    if (x != NULL)
        encode_ok(x);
    return 0;
}
```

We execute a query and encodes the result. Encoding is done in another C module, `pg_encode.c` which is also provided as sample code.

```
static int do_select(const char* s, our_data_t* data, ei_x_buff* x)
{
   PGresult* res = PQexec(data->conn, s);
    encode_result(x, res, data->conn);
    PQclear(res);
    return 0;
}
```

Here we simply checks the result from postgres, and if it's data we encode it as lists of lists with column data. Everything from postgres is C strings, so we just use `ei_x_encode_string` to send the result as strings to Erlang. (The head of the list contains the column names.)

```
void encode_result(ei_x_buff* x, PGresult* res, PGconn* conn)
{
    int row, n_rows, col, n_cols;
    switch (PQresultStatus(res)) {
    case PGRES_TUPLES_OK:
```

```
        n_rows = PQntuples(res);
        n_cols = PQnfields(res);
        ei_x_encode_tuple_header(x, 2);
        encode_ok(x);
        ei_x_encode_list_header(x, n_rows+1);
        ei_x_encode_list_header(x, n_cols);
        for (col = 0; col < n_cols; ++col) {
            ei_x_encode_string(x, PQfname(res, col));
        }
        ei_x_encode_empty_list(x);
        for (row = 0; row < n_rows; ++row) {
            ei_x_encode_list_header(x, n_cols);
            for (col = 0; col < n_cols; ++col) {
                ei_x_encode_string(x, PQgetvalue(res, row, col));
            }
            ei_x_encode_empty_list(x);
        }
        ei_x_encode_empty_list(x);
        break;
    case PGRES_COMMAND_OK:
        ei_x_encode_tuple_header(x, 2);
        encode_ok(x);
        ei_x_encode_string(x, PQcmdTuples(res));
        break;
    default:
        encode_error(x, conn);
        break;
    }
}
```

1.6.3 Compiling and linking the sample driver

The driver should be compiled and linked to a shared library (DLL on windows). With gcc this is done with the link flags `-shared` and `-fpic`. Since we use the `ei` library we should include it too. There are several versions of `ei`, compiled for debug or non-debug and multi-threaded or single-threaded. In the makefile for the samples the `obj` directory is used for the `ei` library, meaning that we use the non-debug, single-threaded version.

1.6.4 Calling a driver as a port in Erlang

Before a driver can be called from Erlang, it must be loaded and opened. Loading is done using the `erl_ddll` module (the `erl_ddll` driver that loads dynamic driver, is actually a driver itself). If loading is ok the port can be opened with `open_port/2`. The port name must match the name of the shared library and the name in the driver entry structure.

When the port has been opened, the driver can be called. In the `pg_sync` example, we don't have any data from the port, only the return value from the `port_control`.

The following code is the Erlang part of the synchronous postgres driver, `pg_sync.erl`.

```
-module(pg_sync).

-define(DRV_CONNECT, 1).
-define(DRV_DISCONNECT, 2).
-define(DRV_SELECT, 3).

-export([connect/1, disconnect/1, select/2]).

connect(ConnectStr) ->
    case erl_ddll:load_driver(".", "pg_sync") of
        ok -> ok;
```

```
        {error, already_loaded} -> ok;
        E -> exit({error, E})
    end,
    Port = open_port({spawn, ?MODULE}, []),
    case binary_to_term(port_control(Port, ?DRV_CONNECT, ConnectStr)) of
        ok -> {ok, Port};
        Error -> Error
    end.

disconnect(Port) ->
    R = binary_to_term(port_control(Port, ?DRV_DISCONNECT, "")),
    port_close(Port),
    R.

select(Port, Query) ->
    binary_to_term(port_control(Port, ?DRV_SELECT, Query)).
```

The api is simple: `connect/1` loads the driver, opens it and logs on to the database, returning the Erlang port if successful, `select/2` sends a query to the driver, and returns the result, `disconnect/1` closes the database connection and the driver. (It does not unload it, however.) The connection string should be a connection string for postgres.

The driver is loaded with `erl_ddll:load_driver/2`, and if this is successful, or if it's already loaded, it is opened. This will call the `start` function in the driver.

We use the `port_control/3` function for all calls into the driver, the result from the driver is returned immediately, and converted to terms by calling `binary_to_term/1`. (We trust that the terms returned from the driver are well-formed, otherwise the `binary_to_term` calls could be contained in a `catch`.)

1.6.5 Sample asynchronous driver

Sometimes database queries can take long time to complete, in our `pg_sync` driver, the emulator halts while the driver is doing it's job. This is often not acceptable, since no other Erlang processes gets a chance to do anything. To improve on our postgres driver, we reimplement it using the asynchronous calls in LibPQ.

The asynchronous version of the driver is in the sample files `pg_async.c` and `pg_asyng.erl`.

```
/* Driver interface declarations */
static ErlDrvData start(ErlDrvPort port, char *command);
static void stop(ErlDrvData drv_data);
static int control(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);
static void ready_io(ErlDrvData drv_data, ErlDrvEvent event);

static ErlDrvEntry pq_driver_entry = {
    NULL,                       /* init */
    start,
    stop,
    NULL,                       /* output */
    ready_io,                   /* ready_input */
    ready_io,                   /* ready_output */
    "pg_async",                 /* the name of the driver */
    NULL,                       /* finish */
    NULL,                       /* handle */
    control,
    NULL,                       /* timeout */
    NULL,                       /* outputv */
    NULL,                       /* ready_async */
    NULL,                       /* flush */
    NULL,                       /* call */
```

```
    NULL                        /* event */
};

typedef struct our_data_t {
    PGconn* conn;
    ErlDrvPort port;
    int socket;
    int connecting;
} our_data_t;
```

Here some things have changed from `pg_sync.c`: we use the entry `ready_io` for `ready_input` and `ready_output` which will be called from the emulator only when there is input to be read from the socket. (Actually, the socket is used in a `select` function inside the emulator, and when the socket is signalled, indicating there is data to read, the `ready_input` entry is called. More on this below.)

Our driver data is also extended, we keep track of the socket used for communication with postgres, and also the port, which is needed when we send data to the port with `driver_output`. We have a flag `connecting` to tell whether the driver is waiting for a connection or waiting for the result of a query. (This is needed since the entry `ready_io` will be called both when connecting and when there is query result.)

```
static int do_connect(const char *s, our_data_t* data)
{
    PGconn* conn = PQconnectStart(s);
    if (PQstatus(conn) == CONNECTION_BAD) {
        ei_x_buff x;
        ei_x_new_with_version(&x);
        encode_error(&x, conn);
        PQfinish(conn);
        conn = NULL;
        driver_output(data->port, x.buff, x.index);
        ei_x_free(&x);
    }
    PQconnectPoll(conn);
    int socket = PQsocket(conn);
    data->socket = socket;
    driver_select(data->port, (ErlDrvEvent)socket, DO_READ, 1);
    driver_select(data->port, (ErlDrvEvent)socket, DO_WRITE, 1);
    data->conn = conn;
    data->connecting = 1;
    return 0;
}
```

The `connect` function looks a bit different too. We connect using the asynchronous `PQconnectStart` function. After the connection is started, we retrieve the socket for the connection with `PQsocket`. This socket is used with the `driver_select` function to wait for connection. When the socket is ready for input or for output, the `ready_io` function will be called.

Note that we only return data (with `driver_output`) if there is an error here, otherwise we wait for the connection to be completed, in which case our `ready_io` function will be called.

```
static int do_select(const char* s, our_data_t* data)
{
    data->connecting = 0;
    PGconn* conn = data->conn;
    /* if there's an error return it now */
    if (PQsendQuery(conn, s) == 0) {
```

```
        ei_x_buff x;
        ei_x_new_with_version(&x);
        encode_error(&x, conn);
        driver_output(data->port, x.buff, x.index);
        ei_x_free(&x);
    }
    /* else wait for ready_output to get results */
    return 0;
}
```

The `do_select` function initiates a select, and returns if there is no immediate error. The actual result will be returned when `ready_io` is called.

```
static void ready_io(ErlDrvData drv_data, ErlDrvEvent event)
{
    PGresult* res = NULL;
    our_data_t* data = (our_data_t*)drv_data;
    PGconn* conn = data->conn;
    ei_x_buff x;
    ei_x_new_with_version(&x);
    if (data->connecting) {
        ConnStatusType status;
        PQconnectPoll(conn);
        status = PQstatus(conn);
        if (status == CONNECTION_OK)
            encode_ok(&x);
        else if (status == CONNECTION_BAD)
            encode_error(&x, conn);
    } else {
        PQconsumeInput(conn);
        if (PQisBusy(conn))
            return;
        res = PQgetResult(conn);
        encode_result(&x, res, conn);
        PQclear(res);
        for (;;) {
            res = PQgetResult(conn);
            if (res == NULL)
                break;
            PQclear(res);
        }
    }
    if (x.index > 1) {
        driver_output(data->port, x.buff, x.index);
        if (data->connecting)
            driver_select(data->port, (ErlDrvEvent)data->socket, DO_WRITE, 0);
    }
    ei_x_free(&x);
}
```

The `ready_io` function will be called when the socket we got from postgres is ready for input or output. Here we first check if we are connecting to the database. In that case we check connection status and return ok if the connection is successful, or error if it's not. If the connection is not yet established, we simply return; `ready_io` will be called again.

If we have result from a connect, indicated that we have data in the `x` buffer, we no longer need to select on output (`ready_output`), so we remove this by calling `driver_select`.

If we're not connecting, we're waiting for results from a `PQsendQuery`, so we get the result and return it. The encoding is done with the same functions as in the earlier example.

We should add error handling here, for instance checking that the socket is still open, but this is just a simple example.

The Erlang part of the asynchronous driver consists of the sample file `pg_async.erl`.

```
-module(pg_async).

-define(DRV_CONNECT, $C).
-define(DRV_DISCONNECT, $D).
-define(DRV_SELECT, $S).

-export([connect/1, disconnect/1, select/2]).

connect(ConnectStr) ->
    case erl_ddll:load_driver(".", "pg_async") of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    Port = open_port({spawn, ?MODULE}, [binary]),
    port_control(Port, ?DRV_CONNECT, ConnectStr),
    case return_port_data(Port) of
        ok ->
            {ok, Port};
        Error ->
            Error
    end.

disconnect(Port) ->
    port_control(Port, ?DRV_DISCONNECT, ""),
    R = return_port_data(Port),
    port_close(Port),
    R.

select(Port, Query) ->
    port_control(Port, ?DRV_SELECT, Query),
    return_port_data(Port).

return_port_data(Port) ->
    receive
        {Port, {data, Data}} ->
            binary_to_term(Data)
    end.
```

The Erlang code is slightly different, this is because we don't return the result synchronously from `port_control`, instead we get it from `driver_output` as data in the message queue. The function `return_port_data` above receives data from the port. Since the data is in binary format, we use `binary_to_term/1` to convert it to Erlang term. Note that the driver is opened in binary mode, `open_port/2` is called with the option `[binary]`. This means that data sent from the driver to the emulator is sent as binaries. Without the `binary` option, they would have been lists of integers.

1.6.6 An asynchronous driver using driver_async

As a final example we demonstrate the use of `driver_async`. We also use the driver term interface. The driver is written in C++. This enables us to use an algorithm from STL. We will use the `next_permutation` algorithm to get the next permutation of a list of integers. For large lists (more than 100000 elements), this will take some time, so we will perform this as an asynchronous task.

The asynchronous api for drivers are quite complicated. First of all, the work must be prepared. In our example we do this in `output`. We could have used `control` just as well, but we want some variation in our examples. In our driver,

we allocate a structure that contains all needed for the asynchronous task to do the work. This is done in the main emulator thread. Then the asynchronous function is called from a driver thread, separate from the main emulator thread. Note that the driver- functions are not reentrant, so they shouldn't be used. Finally, after the function is completed, the driver callback `ready_async` is called from the main emulator thread, this is where we return the result to Erlang. (We can't return the result from within the asynchronous function, since we can't call the driver-functions.)

The code below is from the sample file `next_perm.cc`.

The driver entry looks like before, but also contains the call-back `ready_async`.

```
static ErlDrvEntry next_perm_driver_entry = {
    NULL,                       /* init */
    start,
    NULL,                       /* stop */
    output,
    NULL,                       /* ready_input */
    NULL,                       /* ready_output */
    "next_perm",                /* the name of the driver */
    NULL,                       /* finish */
    NULL,                       /* handle */
    NULL,                       /* control */
    NULL,                       /* timeout */
    NULL,                       /* outputv */
    ready_async,
    NULL,                       /* flush */
    NULL,                       /* call */
    NULL                        /* event */
};
```

The `output` function allocates the work-area of the asynchronous function. Since we use C++, we use a struct, and stuff the data in it. We have to copy the original data, it is not valid after we have returned from the `output` function, and the `do_perm` function will be called later, and from another thread. We return no data here, instead it will be sent later from the `ready_async` call-back.

The `async_data` will be passed to the `do_perm` function. We do not use a `async_free` function (the last argument to `driver_async`, it's only used if the task is cancelled programmatically.

```
struct our_async_data {
    bool prev;
    vector<int> data;
    our_async_data(ErlDrvPort p, int command, const char* buf, int len);
};

our_async_data::our_async_data(ErlDrvPort p, int command,
                               const char* buf, int len)
    : prev(command == 2),
      data((int*)buf, (int*)buf + len / sizeof(int))
{
}

static void do_perm(void* async_data);

static void output(ErlDrvData drv_data, char *buf, int len)
{
    if (*buf < 1 || *buf > 2) return;
    ErlDrvPort port = reinterpret_cast<ErlDrvPort>(drv_data);
    void* async_data = new our_async_data(port, *buf, buf+1, len);
    driver_async(port, NULL, do_perm, async_data, do_free);
```

```
}
```

In the `do_perm` we simply do the work, operating on the structure that was allocated in `output`.

```
static void do_perm(void* async_data)
{
    our_async_data* d = reinterpret_cast<our_async_data*>(async_data);
    if (d->prev)
        prev_permutation(d->data.begin(), d->data.end());
    else
        next_permutation(d->data.begin(), d->data.end());
}
```

In the `ready_async` function, the output is sent back to the emulator. We use the driver term format instead of `ei`. This is the only way to send Erlang terms directly to a driver, without having the Erlang code to call `binary_to_term/1`. In our simple example this works well, and we don't need to use `ei` to handle the binary term format.

When the data is returned we deallocate our data.

```
static void ready_async(ErlDrvData drv_data, ErlDrvThreadData async_data)
{
    ErlDrvPort port = reinterpret_cast<ErlDrvPort>(drv_data);
    our_async_data* d = reinterpret_cast<our_async_data*>(async_data);
    int n = d->data.size(), result_n = n*2 + 3;
    ErlDrvTermData* result = new ErlDrvTermData[result_n], * rp = result;
    for (vector<int>::iterator i = d->data.begin();
         i != d->data.end(); ++i) {
        *rp++ = ERL_DRV_INT;
        *rp++ = *i;
    }
    *rp++ = ERL_DRV_NIL;
    *rp++ = ERL_DRV_LIST;
    *rp++ = n+1;
    driver_output_term(port, result, result_n);
    delete[] result;
    delete d;
}
```

This driver is called like the others from Erlang, however, since we use `driver_output_term`, there is no need to call binary_to_term. The Erlang code is in the sample file `next_perm.erl`.

The input is changed into a list of integers and sent to the driver.

```
-module(next_perm).

-export([next_perm/1, prev_perm/1, load/0, all_perm/1]).

load() ->
    case whereis(next_perm) of
        undefined ->
            case erl_ddll:load_driver(".", "next_perm") of
                ok -> ok;
                {error, already_loaded} -> ok;
```

```
                E -> exit(E)
            end,
            Port = open_port({spawn, "next_perm"}, []),
            register(next_perm, Port);
        _ ->
            ok
    end.

list_to_integer_binaries(L) ->
    [<<I:32/integer-native>> || I <- L].

next_perm(L) ->
    next_perm(L, 1).

prev_perm(L) ->
    next_perm(L, 2).

next_perm(L, Nxt) ->
    load(),
    B = list_to_integer_binaries(L),
    port_control(next_perm, Nxt, B),
    receive
        Result ->
            Result
    end.

all_perm(L) ->
    New = prev_perm(L),
    all_perm(New, L, [New]).

all_perm(L, L, Acc) ->
    Acc;
all_perm(L, Orig, Acc) ->
    New = prev_perm(L),
    all_perm(New, Orig, [New | Acc]).
```

1.7 Inet configuration

1.7.1 Introduction

This chapter tells you how the Erlang runtime system is configured for IP communication. It also explains how you may configure it for your own particular needs by means of a configuration file. The information here is mainly intended for users with special configuration needs or problems. There should normally be no need for specific settings for Erlang to function properly on a correctly IP configured platform.

When Erlang starts up it will read the kernel variable `inetrc` which, if defined, should specify the location and name of a user configuration file. Example:

```
% erl -kernel inetrc '"./cfg_files/erl_inetrc"'
```

Note that the usage of a `.inetrc` file, which was supported in earlier Erlang versions, is now obsolete.

A second way to specify the configuration file is to set the environment variable `ERL_INETRC` to the full name of the file. Example (bash):

```
% export ERL_INETRC=./cfg_files/erl_inetrc
```

Note that the kernel variable `inetrc` overrides this environment variable.

If no user configuration file is specified and Erlang is started in non-distributed or short name distributed mode, Erlang will use default configuration settings and a native lookup method that should work correctly under most

circumstances. Erlang will not read any information from system inet configuration files (like /etc/host.conf, /etc/nsswitch.conf, etc) in these modes, except for /etc/resolv.conf and /etc/hosts that is read and monitored for changes on Unix platforms for the internal DNS client *inet_res*.

If Erlang is started in long name distributed mode, it needs to get the domain name from somewhere and will read system inet configuration files for this information. Any hosts and resolver information found then is also recorded, but not used as long as Erlang is configured for native lookups. (The information becomes useful if the lookup method is changed to `'file'` or `'dns'`, see below).

Native lookup (system calls) is always the default resolver method. This is true for all platforms except VxWorks and OSE Delta where `'file'` or `'dns'` is used (in that order of priority).

On Windows platforms, Erlang will search the system registry rather than look for configuration files when started in long name distributed mode.

1.7.2 Configuration Data

Erlang records the following data in a local database if found in system inet configuration files (or system registry):

- Host names and addresses
- Domain name
- Nameservers
- Search domains
- Lookup method

This data may also be specified explicitly in the user configuration file. The configuration file should contain lines of configuration parameters (each terminated with a full stop). Some parameters add data to the configuration (e.g. host and nameserver), others overwrite any previous settings (e.g. domain and lookup). The user configuration file is always examined last in the configuration process, making it possible for the user to override any default values or previously made settings. Call `inet:get_rc()` to view the state of the inet configuration database.

These are the valid configuration parameters:

`{file, Format, File}.`

```
Format = atom()
File = string()
```

Specify a system file that Erlang should read configuration data from. `Format` tells the parser how the file should be interpreted: `resolv` (Unix resolv.conf), `host_conf_freebsd` (FreeBSD host.conf), `host_conf_bsdos` (BSDOS host.conf), `host_conf_linux` (Linux host.conf), `nsswitch_conf` (Unix nsswitch.conf) or `hosts` (Unix hosts). `File` should specify the name of the file with full path.

`{resolv_conf, File}.`

```
File = string()
```

Specify a system file that Erlang should read resolver configuration from for the internal DNS client *inet_res*, and monitor for changes, even if it does not exist. The path must be absolute.

This may override the configuration parameters `nameserver` and `search` depending on the contents of the specified file. They may also change any time in the future reflecting the file contents.

If the file is specified as an empty string "", no file is read nor monitored in the future. This emulates the old behaviour of not configuring the DNS client when the node is started in short name distributed mode.

If this parameter is not specified it defaults to `/etc/resolv.conf` unless the environment variable `ERL_INET_ETC_DIR` is set which defines the directory for this file to some maybe other than `/etc`.

`{hosts_file, File}.`

```
File = string()
```

Specify a system file that Erlang should read resolver configuration from for the internal hosts file resolver and monitor for changes, even if it does not exist. The path must be absolute.

These host entries are searched after all added with `{file, hosts, File}` above or `{host, IP, Aliases}` below when the lookup option `file` is used.

If the file is specified as an empty string "", no file is read nor monitored in the future. This emulates the old behaviour of not configuring the DNS client when the node is started in short name distributed mode.

If this parameter is not specified it defaults to `/etc/hosts` unless the environment variable `ERL_INET_ETC_DIR` is set which defines the directory for this file to some maybe other than `/etc`.

`{registry, Type}.`

```
Type = atom()
```

Specify a system registry that Erlang should read configuration data from. Currently, `win32` is the only valid option.

`{host, IP, Aliases}.`

```
IP = tuple()
Aliases = [string()]
```

Add host entry to the hosts table.

`{domain, Domain}.`

```
Domain = string()
```

Set domain name.

`{nameserver, IP [,Port]}.`

```
IP = tuple()
Port = integer()
```

Add address (and port, if other than default) of primary nameserver to use for *inet_res*.

`{alt_nameserver, IP [,Port]}.`

```
IP = tuple()
Port = integer()
```

Add address (and port, if other than default) of secondary nameserver for *inet_res*.

`{search, Domains}.`

```
Domains = [string()]
```

Add search domains for *inet_res*.

{lookup, Methods}.

```
Methods = [atom()]
```

Specify lookup methods and in which order to try them. The valid methods are: `native` (use system calls), `file` (use host data retrieved from system configuration files and/or the user configuration file) or `dns` (use the Erlang DNS client *inet_res* for nameserver queries).

The lookup method `string` tries to parse the hostname as a IPv4 or IPv6 string and return the resulting IP address. It is automatically tried first when `native` is *not* in the `Methods` list. To skip it in this case the pseudo lookup method `nostring` can be inserted anywhere in the `Methods` list.

{cache_size, Size}.

```
Size = integer()
```

Set size of resolver cache. Default is 100 DNS records.

{cache_refresh, Time}.

```
Time = integer()
```

Set how often (in millisec) the resolver cache for *inet_res*. is refreshed (i.e. expired DNS records are deleted). Default is 1 h.

{timeout, Time}.

```
Time = integer()
```

Set the time to wait until retry (in millisec) for DNS queries made by *inet_res*. Default is 2 sec.

{retry, N}.

```
N = integer()
```

Set the number of DNS queries *inet_res* will try before giving up. Default is 3.

{inet6, Bool}.

```
Bool = true | false
```

Tells the DNS client *inet_res* to look up IPv6 addresses. Default is false.

{usevc, Bool}.

```
Bool = true | false
```

Tells the DNS client *inet_res* to use TCP (Virtual Circuit) instead of UDP. Default is false.

{edns, Version}.

```
Version = false | 0
```

Sets the EDNS version that *inet_res* will use. The only allowed is zero. Default is false which means to not use EDNS.

{udp_payload_size, Size}.

```
N = integer()
```

Sets the allowed UDP payload size *inet_res* will advertise in EDNS queries. Also sets the limit when the DNS query will be deemed too large for UDP forcing a TCP query instead, which is not entirely correct since the advertised UDP payload size of the individual nameserver is what should be used, but this simple strategy will do until a more intelligent (probing, caching) algorithm need be implemented. The default is 1280 which stems from the standard Ethernet MTU size.

{udp, Module}.

```
Module = atom()
```

Tell Erlang to use other primitive UDP module than inet_udp.

{tcp, Module}.

```
Module = atom()
```

Tell Erlang to use other primitive TCP module than inet_tcp.

clear_hosts.

Clear the hosts table.

clear_ns.

Clear the list of recorded nameservers (primary and secondary).

clear_search.

Clear the list of search domains.

1.7.3 User Configuration Example

Here follows a user configuration example.

Assume a user does not want Erlang to use the native lookup method, but wants Erlang to read all information necessary from start and use that for resolving names and addresses. In case lookup fails, Erlang should request the data from a nameserver (using the Erlang DNS client, set to use EDNS allowing larger responses). The resolver configuration will be updated when its configuration file changes, furthermore, DNS records should never be cached. The user configuration file (in this example named `erl_inetrc`, stored in directory `./cfg_files`) could then look like this (Unix):

```
%% -- ERLANG INET CONFIGURATION FILE --
%% read the hosts file
{file, hosts, "/etc/hosts"}.
%% add a particular host
{host, {134,138,177,105}, ["finwe"]}.
%% do not monitor the hosts file
{hosts_file, ""}.
```

```
%% read and monitor nameserver config from here
{resolv_conf, "/usr/local/etc/resolv.conf"}.
%% enable EDNS
{edns,0}.
%% disable caching
{cache_size, 0}.
%% specify lookup method
{lookup, [file, dns]}.
```

And Erlang could, for example, be started like this:

```
% erl -sname my_node -kernel inetrc '"./cfg_files/erl_inetrc"'
```

1.8 External Term Format

1.8.1 Introduction

The external term format is mainly used in the distribution mechanism of Erlang.

Since Erlang has a fixed number of types, there is no need for a programmer to define a specification for the external format used within some application. All Erlang terms has an external representation and the interpretation of the different terms are application specific.

In Erlang the BIF *term_to_binary/1,2* is used to convert a term into the external format. To convert binary data encoding a term the BIF *binary_to_term/1* is used.

The distribution does this implicitly when sending messages across node boundaries.

The overall format of the term format is:

1	1	N
`131`	`Tag`	`Data`

Table 8.1:

> ### Note:
>
> When messages are *passed between connected nodes* and a *distribution header* is used, the first byte containing the version number (131) is omitted from the terms that follow the distribution header. This since the version number is implied by the version number in the distribution header.

A compressed term looks like this:

1	1	4	N
131	80	UncompressedSize	Zlib-compressedData

Table 8.2:

Uncompressed Size (unsigned 32 bit integer in big-endian byte order) is the size of the data before it was compressed. The compressed data has the following format when it has been expanded:

1	Uncompressed Size
Tag	Data

Table 8.3:

1.8.2 Distribution header

As of erts version 5.7.2 the old atom cache protocol was dropped and a new one was introduced. This atom cache protocol introduced the distribution header. Nodes with erts versions earlier than 5.7.2 can still communicate with new nodes, but no distribution header and no atom cache will be used.

The distribution header currently only contains an atom cache reference section, but could in the future contain more information. The distribution header precedes one or more Erlang terms on the external format. For more information see the documentation of the *protocol between connected nodes* in the *distribution protocol* documentation.

ATOM_CACHE_REF entries with corresponding `AtomCacheReferenceIndex` in terms encoded on the external format following a distribution header refers to the atom cache references made in the distribution header. The range is 0 <= `AtomCacheReferenceIndex` < 255, i.e., at most 255 different atom cache references from the following terms can be made.

The distribution header format is:

1	1	1	NumberOfAtomCacheRefs/2+1 | 0	N | 0
131	68	NumberOfAtomCacheRefs	Flags	AtomCacheRefs

Table 8.4:

`Flags` consists of `NumberOfAtomCacheRefs/2+1` bytes, unless `NumberOfAtomCacheRefs` is `0`. If `NumberOfAtomCacheRefs` is `0`, `Flags` and `AtomCacheRefs` are omitted. Each atom cache reference have a half byte flag field. Flags corresponding to a specific `AtomCacheReferenceIndex`, are located in flag byte number `AtomCacheReferenceIndex/2`. Flag byte 0 is the first byte after the `NumberOfAtomCacheRefs` byte. Flags for an even `AtomCacheReferenceIndex` are located in the least significant half byte and flags for an odd `AtomCacheReferenceIndex` are located in the most significant half byte.

The flag field of an atom cache reference has the following format:

1 bit	3 bits
NewCacheEntryFlag	SegmentIndex

Table 8.5:

The most significant bit is the `NewCacheEntryFlag`. If set, the corresponding cache reference is new. The three least significant bits are the `SegmentIndex` of the corresponding atom cache entry. An atom cache consists of 8 segments each of size 256, i.e., an atom cache can contain 2048 entries.

After flag fields for atom cache references, another half byte flag field is located which has the following format:

3 bits	1 bit
CurrentlyUnused	LongAtoms

Table 8.6:

The least significant bit in that half byte is the `LongAtoms` flag. If it is set, 2 bytes are used for atom lengths instead of 1 byte in the distribution header. However, the current emulator cannot handle long atoms, so it will currently always be 0.

After the `Flags` field follow the `AtomCacheRefs`. The first `AtomCacheRef` is the one corresponding to `AtomCacheReferenceIndex` 0. Higher indices follows in sequence up to index `NumberOfAtomCacheRefs - 1`.

If the `NewCacheEntryFlag` for the next `AtomCacheRef` has been set, a `NewAtomCacheRef` on the following format will follow:

1	1 | 2	Length
InternalSegmentIndex	Length	AtomText

Table 8.7:

`InternalSegmentIndex` together with the `SegmentIndex` completely identify the location of an atom cache entry in the atom cache. `Length` is number of one byte characters that the atom text consists of. Length is a two byte big endian integer if the `LongAtoms` flag has been set, otherwise a one byte integer. Subsequent `CachedAtomRef`s with the same `SegmentIndex` and `InternalSegmentIndex` as this `NewAtomCacheRef` will refer to this atom until a new `NewAtomCacheRef` with the same `SegmentIndex` and `InternalSegmentIndex` appear.

If the `NewCacheEntryFlag` for the next `AtomCacheRef` has not been set, a `CachedAtomRef` on the following format will follow:

1
InternalSegmentIndex

Table 8.8:

`InternalSegmentIndex` together with the `SegmentIndex` identify the location of the atom cache entry in the atom cache. The atom corresponding to this `CachedAtomRef` is the latest `NewAtomCacheRef` preceding this `CachedAtomRef` in another previously passed distribution header.

1.8.3 ATOM_CACHE_REF

1	1
82	AtomCacheReferenceIndex

Table 8.9:

Refers to the atom with `AtomCacheReferenceIndex` in the *distribution header*.

1.8.4 SMALL_INTEGER_EXT

1	1
97	Int

Table 8.10:

Unsigned 8 bit integer.

1.8.5 INTEGER_EXT

1	4
98	Int

Table 8.11:

Signed 32 bit integer in big-endian format (i.e. MSB first)

1.8.6 FLOAT_EXT

1	31
99	Float String

Table 8.12:

A float is stored in string format. the format used in sprintf to format the float is "%.20e" (there are more bytes allocated than necessary). To unpack the float use sscanf with format "%lf".

This term is used in minor version 0 of the external format; it has been superseded by *NEW_FLOAT_EXT* .

1.8.7 ATOM_EXT

1	2	Len
`100`	`Len`	`AtomName`

Table 8.13:

An atom is stored with a 2 byte unsigned length in big-endian order, followed by `Len` numbers of 8 bit characters that forms the `AtomName`. Note: The maximum allowed value for `Len` is 255.

1.8.8 REFERENCE_EXT

1	N	4	1
101	Node	ID	Creation

Table 8.14:

Encode a reference object (an object generated with `make_ref/0`). The `Node` term is an encoded atom, i.e. *ATOM_EXT*, *SMALL_ATOM_EXT* or *ATOM_CACHE_REF*. The `ID` field contains a big-endian unsigned integer, but *should be regarded as uninterpreted data* since this field is node specific. `Creation` is a byte containing a node serial number that makes it possible to separate old (crashed) nodes from a new one.

In `ID`, only 18 bits are significant; the rest should be 0. In `Creation`, only 2 bits are significant; the rest should be 0. See *NEW_REFERENCE_EXT*.

1.8.9 PORT_EXT

1	N	4	1
102	Node	ID	Creation

Table 8.15:

Encode a port object (obtained form `open_port/2`). The `ID` is a node specific identifier for a local port. Port operations are not allowed across node boundaries. The `Creation` works just like in *REFERENCE_EXT*.

1.8.10 PID_EXT

1	N	4	4	1
103	Node	ID	Serial	Creation

Table 8.16:

Encode a process identifier object (obtained from `spawn/3` or friends). The `ID` and `Creation` fields works just like in *REFERENCE_EXT*, while the `Serial` field is used to improve safety. In `ID`, only 15 bits are significant; the rest should be 0.

1.8.11 SMALL_TUPLE_EXT

1	1	N
104	Arity	Elements

Table 8.17:

SMALL_TUPLE_EXT encodes a tuple. The Arity field is an unsigned byte that determines how many element that follows in the Elements section.

1.8.12 LARGE_TUPLE_EXT

1	4	N
105	Arity	Elements

Table 8.18:

Same as *SMALL_TUPLE_EXT* with the exception that Arity is an unsigned 4 byte integer in big endian format.

1.8.13 NIL_EXT

1
106

Table 8.19:

The representation for an empty list, i.e. the Erlang syntax [].

1.8.14 STRING_EXT

1	2	Len
107	Length	Characters

Table 8.20:

String does NOT have a corresponding Erlang representation, but is an optimization for sending lists of bytes (integer in the range 0-255) more efficiently over the distribution. Since the Length field is an unsigned 2 byte integer (big endian), implementations must make sure that lists longer than 65535 elements are encoded as *LIST_EXT*.

1.8.15 LIST_EXT

1	4		
108	Length	Elements	Tail

Table 8.21:

Length is the number of elements that follows in the Elements section. Tail is the final tail of the list; it is *NIL_EXT* for a proper list, but may be anything type if the list is improper (for instance [a|b]).

1.8.16 BINARY_EXT

1	4	Len
109	Len	Data

Table 8.22:

Binaries are generated with bit syntax expression or with *list_to_binary/1*, *term_to_binary/1*, or as input from binary ports. The `Len` length field is an unsigned 4 byte integer (big endian).

1.8.17 SMALL_BIG_EXT

1	1	1	n
110	n	Sign	d(0) ... d(n-1)

Table 8.23:

Bignums are stored in unary form with a `Sign` byte that is 0 if the binum is positive and 1 if is negative. The digits are stored with the LSB byte stored first. To calculate the integer the following formula can be used:
B = 256
(d0*B^0 + d1*B^1 + d2*B^2 + ... d(N-1)*B^(n-1))

1.8.18 LARGE_BIG_EXT

1	4	1	n
111	n	Sign	d(0) ... d(n-1)

Table 8.24:

Same as *SMALL_BIG_EXT* with the difference that the length field is an unsigned 4 byte integer.

1.8.19 NEW_REFERENCE_EXT

1	2	N	1	N'
114	Len	Node	Creation	ID ...

Table 8.25:

Node and Creation are as in *REFERENCE_EXT*.

`ID` contains a sequence of big-endian unsigned integers (4 bytes each, so `N'` is a multiple of 4), but should be regarded as uninterpreted data.

`N'` = 4 * `Len`.

In the first word (four bytes) of `ID`, only 18 bits are significant, the rest should be 0. In `Creation`, only 2 bits are significant, the rest should be 0.

NEW_REFERENCE_EXT was introduced with distribution version 4. In version 4, `N'` should be at most 12.

See *REFERENCE_EXT*).

1.8.20 SMALL_ATOM_EXT

1	1	Len
`115`	`Len`	`AtomName`

Table 8.26:

An atom is stored with a 1 byte unsigned length, followed by `Len` numbers of 8 bit characters that forms the `AtomName`. Longer atoms can be represented by *ATOM_EXT*. *Note* the `SMALL_ATOM_EXT` was introduced in erts version 5.7.2 and require a small atom distribution flag exchanged in the distribution handshake.

1.8.21 FUN_EXT

1	4	N1	N2	N3	N4	N5
117	NumFree	Pid	Module	Index	Uniq	Free vars ...

Table 8.27:

`Pid`
: is a process identifier as in *PID_EXT*. It represents the process in which the fun was created.

`Module`
: is an encoded as an atom, using *ATOM_EXT*, *SMALL_ATOM_EXT* or *ATOM_CACHE_REF*. This is the module that the fun is implemented in.

`Index`
: is an integer encoded using *SMALL_INTEGER_EXT* or *INTEGER_EXT*. It is typically a small index into the module's fun table.

`Uniq`
: is an integer encoded using *SMALL_INTEGER_EXT* or *INTEGER_EXT*. `Uniq` is the hash value of the parse for the fun.

`Free vars`
: is `NumFree` number of terms, each one encoded according to its type.

1.8.22 NEW_FUN_EXT

1	4	1	16	4	4	N1	N2	N3	N4	N5
112	Size	Arity	Uniq	Index	NumFree	Module	OldIndex	OldUniq	Pid	Free Vars

Table 8.28:

This is the new encoding of internal funs: `fun F/A` and `fun(Arg1,..) -> ... end`.

`Size`
is the total number of bytes, including the `Size` field.

`Arity`
is the arity of the function implementing the fun.

`Uniq`
is the 16 bytes MD5 of the significant parts of the Beam file.

`Index`
is an index number. Each fun within a module has an unique index. `Index` is stored in big-endian byte order.

`NumFree`
is the number of free variables.

`Module`
is an encoded as an atom, using *ATOM_EXT*, *SMALL_ATOM_EXT* or *ATOM_CACHE_REF*. This is the module that the fun is implemented in.

`OldIndex`
is an integer encoded using *SMALL_INTEGER_EXT* or *INTEGER_EXT*. It is typically a small index into the module's fun table.

`OldUniq`
is an integer encoded using *SMALL_INTEGER_EXT* or *INTEGER_EXT*. `Uniq` is the hash value of the parse tree for the fun.

`Pid`
is a process identifier as in *PID_EXT*. It represents the process in which the fun was created.

`Free vars`
is `NumFree` number of terms, each one encoded according to its type.

1.8.23 EXPORT_EXT

1	N1	N2	N3
113	Module	Function	Arity

Table 8.29:

This term is the encoding for external funs: `fun M:F/A`.

`Module` and `Function` are atoms (encoded using *ATOM_EXT*, *SMALL_ATOM_EXT* or *ATOM_CACHE_REF*).

`Arity` is an integer encoded using *SMALL_INTEGER_EXT*.

1.8.24 BIT_BINARY_EXT

1	4	1	Len
77	Len	Bits	Data

Table 8.30:

This term represents a bitstring whose length in bits is not a multiple of 8 (created using the bit syntax in R12B and later). The `Len` field is an unsigned 4 byte integer (big endian). The `Bits` field is the number of bits that are used in the last byte in the data field, counting from the most significant bit towards the least significant.

1.8.25 NEW_FLOAT_EXT

1	8
70	IEEE float

Table 8.31:

A float is stored as 8 bytes in big-endian IEEE format.

This term is used in minor version 1 of the external format.

1.9 Distribution Protocol

The description here is far from complete and will therefore be further refined in upcoming releases. The protocols both from Erlang nodes towards EPMD (Erlang Port Mapper Daemon) and between Erlang nodes, however, are stable since many years.

The distribution protocol can be divided into four (4) parts:

- 1. Low level socket connection.
- 2. Handshake, interchange node name and authenticate.
- 3. Authentication (done by net_kernel).
- 4. Connected.

A node fetches the Port number of another node through the EPMD (at the other host) in order to initiate a connection request.

For each host where a distributed Erlang node is running there should also be an EPMD running. The EPMD can be started explicitly or automatically as a result of the Erlang node startup.

By default EPMD listens on port 4369.

3 and 4 are performed at the same level but the net_kernel disconnects the other node if it communicates using an invalid cookie (after one (1) second).

The integers in all multi-byte fields are in big-endian order.

1.9.1 EPMD Protocol

The requests served by the EPMD (Erlang Port Mapper Daemon) are summarized in the figure below.

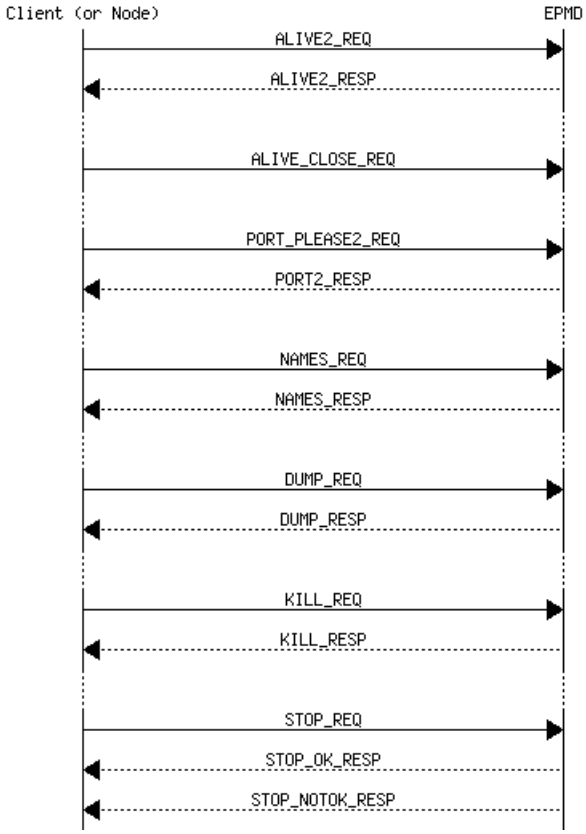

Figure 9.1: Summary of EPMD requests.

Each request `*_REQ` is preceded by a two-byte length field. Thus, the overall request format is:

2	n
Length	Request

Table 9.1:

Register a node in the EPMD

When a distributed node is started it registers itself in EPMD. The message ALIVE2_REQ described below is sent from the node towards EPMD. The response from EPMD is ALIVE2_RESP.

1	2	1	1	2	2	2	Nlen	2	Elen
120	PortNo	NodeType	Protocol	HighestVersion	LowestVersion	Nlen	NodeName	Elen	Extra

Table 9.2: ALIVE2_REQ (120)

`PortNo`
The port number on which the node accept connection requests.

`NodeType`
77 = normal Erlang node, 72 = hidden node (C-node),...
`Protocol`
0 = tcp/ip-v4, ...
`HighestVersion`
The highest distribution version that this node can handle. The value in R6B and later is 5.
`LowestVersion`
The lowest distribution version that this node can handle. The value in R6B and later is 5.
`Nlen`
The length of the `NodeName`.
`NodeName`
The NodeName as a string of length `Nlen`.
`Elen`
The length of the `Extra` field.
`Extra`
Extra field of `Elen` bytes.

The connection created to the EPMD must be kept as long as the node is a distributed node. When the connection is closed the node is automatically unregistered from the EPMD.

The response message ALIVE2_RESP is described below.

1	1	2
121	Result	Creation

Table 9.3: ALIVE2_RESP (121)

Result = 0 -> ok, Result > 0 -> error

Unregister a node from the EPMD

A node unregister itself from the EPMD by simply closing the TCP connection towards EPMD established when the node was registered.

Get the distribution port of another node

When one node wants to connect to another node it starts with a PORT_PLEASE2_REQ request towards EPMD on the host where the node resides in order to get the distribution port that the node listens to.

1	N
122	NodeName

Table 9.4: PORT_PLEASE2_REQ (122)

where N = Length - 1

1	1

119	Result

Table 9.5: PORT2_RESP (119) response indicating error, Result > 0.

Or

1	1	2	1	1	2	2	2	Nlen	2	Elen
119	Result	PortNo	NodeType	Protocol	HighestVersion	LowestVersion	Nlen	NodeName	Elen	Extra

Table 9.6: PORT2_RESP when Result = 0.

If Result > 0, the packet only consists of [119, Result].

EPMD will close the socket as soon as it has sent the information.

Get all registered names from EPMD

This request is used via the Erlang function `net_adm:names/1,2`. A TCP connection is opened towards EPMD and this request is sent.

1
110

Table 9.7: NAMES_REQ (110)

The response for a `NAMES_REQ` looks like this:

4	
EPMDPortNo	NodeInfo*

Table 9.8: NAMES_RESP

NodeInfo is a string written for each active node. When all NodeInfo has been written the connection is closed by EPMD.

NodeInfo is, as expressed in Erlang:

```
io:format("name ~s at port ~p~n", [NodeName, Port]).
```

Dump all data from EPMD

This request is not really used, it should be regarded as a debug feature.

1

100

Table 9.9: DUMP_REQ

The response for a `DUMP_REQ` looks like this:

4	
EPMDPortNo	NodeInfo*

Table 9.10: DUMP_RESP

NodeInfo is a string written for each node kept in EPMD. When all NodeInfo has been written the connection is closed by EPMD.

NodeInfo is, as expressed in Erlang:

```
io:format("active name     ~s at port ~p, fd = ~p ~n",
          [NodeName, Port, Fd]).
```

or

```
io:format("old/unused name ~s at port ~p, fd = ~p~n",
          [NodeName, Port, Fd]).
```

Kill the EPMD

This request will kill the running EPMD. It is almost never used.

1
107

Table 9.11: KILL_REQ

The response fo a `KILL_REQ` looks like this:

2
OKString

Table 9.12: KILL_RESP

where `OKString` is "OK".

STOP_REQ (Not Used)

1	n
115	NodeName

Table 9.13: STOP_REQ

where n = Length - 1

The current implementation of Erlang does not care if the connection to the EPMD is broken.

The response for a `STOP_REQ` looks like this.

7
OKString

Table 9.14: STOP_RESP

where OKString is "STOPPED".

A negative response can look like this.

7
NOKString

Table 9.15: STOP_NOTOK_RESP

where NOKString is "NOEXIST".

1.9.2 Handshake

The handshake is discussed in detail in the internal documentation for the kernel (Erlang) application.

1.9.3 Protocol between connected nodes

As of erts version 5.7.2 the runtime system passes a distribution flag in the handshake stage that enables the use of a *distribution header* on all messages passed. Messages passed between nodes are in this case on the following format:

4	d	n	m
`Length`	`DistributionHeader`	`ControlMessage`	`Message`

Table 9.16:

where:

`Length` is equal to d + n + m

`ControlMessage` is a tuple passed using the external format of Erlang.

`Message` is the message sent to another node using the '!' (in external format). Note that `Message` is only passed in combination with a `ControlMessage` encoding a send ('!').

Also note that *the version number is omitted from the terms that follow a distribution header*.

Nodes with an erts version less than 5.7.2 does not pass the distribution flag that enables the distribution header. Messages passed between nodes are in this case on the following format:

4	1	n	m
`Length`	`Type`	`ControlMessage`	`Message`

Table 9.17:

where:

`Length` is equal to 1 + n + m

Type is: 112 (pass through)

`ControlMessage` is a tuple passed using the external format of Erlang.

`Message` is the message sent to another node using the '!' (in external format). Note that `Message` is only passed in combination with a `ControlMessage` encoding a send ('!').

The `ControlMessage` is a tuple, where the first element indicates which distributed operation it encodes.

`LINK`

```
{1, FromPid, ToPid}
```

`SEND`

```
{2, Cookie, ToPid}
```

Note followed by `Message`

`EXIT`

```
{3, FromPid, ToPid, Reason}
```

`UNLINK`

```
{4, FromPid, ToPid}
```

`NODE_LINK`

```
{5}
```

`REG_SEND`

```
{6, FromPid, Cookie, ToName}
```

Note followed by `Message`

`GROUP_LEADER`

```
{7, FromPid, ToPid}
```

`EXIT2`

```
{8, FromPid, ToPid, Reason}
```

1.9.4 New Ctrlmessages for distrvsn = 1 (OTP R4)

`SEND_TT`

`{12, Cookie, ToPid, TraceToken}`

Note followed by `Message`

`EXIT_TT`

`{13, FromPid, ToPid, TraceToken, Reason}`

`REG_SEND_TT`

`{16, FromPid, Cookie, ToName, TraceToken}`

Note followed by `Message`

`EXIT2_TT`

`{18, FromPid, ToPid, TraceToken, Reason}`

1.9.5 New Ctrlmessages for distrvsn = 2

distrvsn 2 was never used.

1.9.6 New Ctrlmessages for distrvsn = 3 (OTP R5C)

None, but the version number was increased anyway.

1.9.7 New Ctrlmessages for distrvsn = 4 (OTP R6)

These are only recognized by Erlang nodes, not by hidden nodes.

`MONITOR_P`

`{19, FromPid, ToProc, Ref}` `FromPid` = monitoring process `ToProc` = monitored process pid or name (atom)

`DEMONITOR_P`

`{20, FromPid, ToProc, Ref}` We include the FromPid just in case we want to trace this. `FromPid` = monitoring process `ToProc` = monitored process pid or name (atom)

`MONITOR_P_EXIT`

`{21, FromProc, ToPid, Ref, Reason}` `FromProc` = monitored process pid or name (atom) `ToPid` = monitoring process `Reason` = exit reason for the monitored process

2 Reference Manual

The Erlang Runtime System Application *ERTS*.

> **Note:**
>
> By default, the `erts` is only guaranteed to be compatible with other Erlang/OTP components from the same release as the `erts` itself. See the documentation of the system flag *+R* on how to communicate with Erlang/OTP components from earlier releases.

erl_prim_loader

Erlang module

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

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

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

> ### Warning:
>
> The support for loading of code from archive files is experimental. The sole purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces etc. may be changed in a future release. The functions `list_dir/1` and `read_file_info/1` as well as the flag `-loader_debug` are also experimental

Exports

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

Types:

Id = term()

Loader = atom() | string()

Hosts = [Host]

Host = atom()

Pid = pid()

What = term()

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

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

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

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

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

Types:

Filename = string()

Bin = binary()

FullName = string()

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

The `Filename` can also be a file in an archive. For example `/otp/root/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin/mnesia_backup.beam` See *code(3)* about archive files.

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

Types:

Path = [Dir]

Dir = string()

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

`list_dir(Dir) -> {ok, Filenames} | error`

Types:

Dir = name()

Filenames = [Filename]

Filename = string()

Lists all the files in a directory. Returns `{ok, Filenames}` if successful. Otherwise, it returns `error`. `Filenames` is a list of the names of all the files in the directory. The names are not sorted.

The `Dir` can also be a directory in an archive. For example `/otp/root/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin` See *code(3)* about archive files.

`read_file_info(Filename) -> {ok, FileInfo} | error`

Types:

Filename = name()

FileInfo = #file_info{}

Retrieves information about a file. Returns `{ok, FileInfo}` if successful, otherwise `error`. `FileInfo` is a record `file_info`, defined in the Kernel include file `file.hrl`. Include the following directive in the module from which the function is called:

```
-include_lib("kernel/include/file.hrl").
```

See *file(3)* for more info about the record `file_info`.

The `Filename` can also be a file in an archive. For example `/otp/root/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin/mnesia_backup.beam` See *code(3)* about archive files.

`set_path(Path) -> ok`

Types:

Path = [Dir]

Dir = string()

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

Protocol

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

```
Function          Send               Receive
------------------------------------------------------------
get_file          [102 | FileName]   [121 | BinaryFile] (on success)
                                     [122]              (failure)

stop              eof                terminate
```

Command Line Flags

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

`-loader Loader`

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

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

`-loader_debug`

Makes the `efile` loader write some debug information, such as the reason for failures, while it handles files.

`-hosts Hosts`

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

`-id Id`

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

`-setcookie Cookie`

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

SEE ALSO

init(3), erl_boot_server(3)

erlang

Erlang module

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

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

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

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

DATA TYPES

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

iodata() = iolist() | binary()

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

Exports

`abs(Number) -> int() | float()`

Types:

Number = number()

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

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

Allowed in guard tests.

`adler32(Data) -> int()`

Types:

Data = iodata()

Computes and returns the adler32 checksum for `Data`.

`adler32(OldAdler, Data) -> int()`

Types:

OldAdler = int()

Data = iodata()

Continue computing the adler32 checksum by combining the previous checksum, `OldAdler`, with the checksum of `Data`.

The following code:

```
X = adler32(Data1),
Y = adler32(X,Data2).
```

- would assign the same value to `Y` as this would:

```
Y = adler32([Data1,Data2]).
```

`adler32_combine(FirstAdler, SecondAdler, SecondSize) -> int()`

Types:

FirstAdler = SecondAdler = int()

SecondSize = int()

Combines two previously computed adler32 checksums. This computation requires the size of the data object for the second checksum to be known.

The following code:

```
Y = adler32(Data1),
Z = adler32(Y,Data2).
```

- would assign the same value to `Z` as this would:

```
X = adler32(Data1),
Y = adler32(Data2),
Z = adler32_combine(X,Y,iolist_size(Data2)).
```

`erlang:append_element(Tuple1, Term) -> Tuple2`

Types:

Tuple1 = Tuple2 = tuple()

Term = term()

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

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

```
{one,two,three}
```

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

Types:

Fun = fun()

Args = [term()]

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

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

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

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

Types:

Module = Function = atom()

Args = [term()]

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

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

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

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

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

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

atom_to_binary(Atom, Encoding) -> binary()

Types:

Atom = atom()

Encoding = latin1 | utf8 | unicode

Returns a binary which corresponds to the text representation of `Atom`. If `Encoding` is `latin1`, there will be one byte for each character in the text representation. If `Encoding` is `utf8` or `unicode`, the characters will encoded using UTF-8 (meaning that characters from 16#80 up to 0xFF will be encode in two bytes).

> Note:
>
> Currently, `atom_to_binary(Atom, latin1)` can never fail because the text representation of an atom can only contain characters from 0 to 16#FF. In a future release, the text representation of atoms might be allowed to contain any Unicode character and `atom_to_binary(Atom, latin1)` will fail if the text representation for the `Atom` contains a Unicode character greater than 16#FF.

```
> atom_to_binary('Erlang', latin1).
<<"Erlang">>
```

`atom_to_list(Atom) -> string()`

Types:

Atom = atom()

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

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

`binary_to_atom(Binary, Encoding) -> atom()`

Types:

Binary = binary()

Encoding = latin1 | utf8 | unicode

Returns the atom whose text representation is `Binary`. If `Encoding` is `latin1`, no translation of bytes in the binary is done. If `Encoding` is `utf8` or `unicode`, the binary must contain valid UTF-8 sequences; furthermore, only Unicode characters up to 0xFF are allowed.

> Note:
>
> `binary_to_atom(Binary, utf8)` will fail if the binary contains Unicode characters greater than 16#FF. In a future release, such Unicode characters might be allowed and `binary_to_atom(Binary, utf8)` will not fail in that case.

```
> binary_to_atom(<<"Erlang">>, latin1).
'Erlang'
> binary_to_atom(<<1024/utf8>>, utf8).
** exception error: bad argument
     in function  binary_to_atom/2
        called as binary_to_atom(<<208,128>>,utf8)
```

`binary_to_existing_atom(Binary, Encoding) -> atom()`

Types:

Binary = binary()

Encoding = latin1 | utf8 | unicode

Works like *binary_to_atom/2*, but the atom must already exist.

Failure: `badarg` if the atom does not already exist.

`binary_to_list(Binary) -> [char()]`

Types:

Binary = binary()

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

`binary_to_list(Binary, Start, Stop) -> [char()]`

Types:

Binary = binary()

Start = Stop = 1..byte_size(Binary)

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

`bitstring_to_list(Bitstring) -> [char()|bitstring()]`

Types:

Bitstring = bitstring()

Returns a list of integers which correspond to the bytes of `Bitstring`. If the number of bits in the binary is not divisible by 8, the last element of the list will be a bitstring containing the remaining bits (1 up to 7 bits).

`binary_to_term(Binary) -> term()`

Types:

Binary = ext_binary()

Returns an Erlang term which is the result of decoding the binary object `Binary`, which must be encoded according to the Erlang external term format.

> Warning:
>
> When decoding binaries from untrusted sources, consider using `binary_to_term/2` to prevent denial of service attacks.

See also *term_to_binary/1* and *binary_to_term/2*.

`erlang:binary_to_term(Binary, Opts) -> term()`

Types:

Opts = [safe]

Binary = ext_binary()

As `binary_to_term/1`, but takes options that affect decoding of the binary.

`safe`

Use this option when receiving binaries from an untrusted source.

When enabled, it prevents decoding data that may be used to attack the Erlang system. In the event of receiving unsafe data, decoding fails with a badarg error.

Currently, this prevents creation of new atoms directly, creation of new atoms indirectly (as they are embedded in certain structures like pids, refs, funs, etc.), and creation of new external function references. None of those resources are currently garbage collected, so unchecked creation of them can exhaust available memory.

Failure: `badarg` if `safe` is specified and unsafe data is decoded.

See also *term_to_binary/1*, *binary_to_term/1*, and *list_to_existing_atom/1*.

bit_size(Bitstring) -> int()

Types:

Bitstring = bitstring()

Returns an integer which is the size in bits of `Bitstring`.

```
> bit_size(<<433:16,3:3>>).
19
> bit_size(<<1,2,3>>).
24
```

Allowed in guard tests.

erlang:bump_reductions(Reductions) -> void()

Types:

Reductions = int()

This implementation-dependent function increments the reduction counter for the calling process. In the Beam emulator, the reduction counter is normally incremented by one for each function and BIF call, and a context switch is forced when the counter reaches the maximum number of reductions for a process (2000 reductions in R12B).

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

byte_size(Bitstring) -> int()

Types:

Bitstring = bitstring()

Returns an integer which is the number of bytes needed to contain `Bitstring`. (That is, if the number of bits in `Bitstring` is not divisible by 8, the resulting number of bytes will be rounded *up*.)

```
> byte_size(<<433:16,3:3>>).
3
> byte_size(<<1,2,3>>).
3
```

Allowed in guard tests.

erlang:cancel_timer(TimerRef) -> Time | false

Types:

TimerRef = ref()

Time = int()

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

See also *erlang:send_after/3*, *erlang:start_timer/3*, and *erlang:read_timer/1*.

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

check_process_code(Pid, Module) -> bool()

Types:

Pid = pid()

Module = atom()

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

```
> check_process_code(Pid, lists).
false
```

See also *code(3)*.

concat_binary(ListOfBinaries)

Do not use; use *list_to_binary/1* instead.

crc32(Data) -> int()

Types:

Data = iodata()

Computes and returns the crc32 (IEEE 802.3 style) checksum for `Data`.

crc32(OldCrc, Data) -> int()

Types:

OldCrc = int()

Data = iodata()

Continue computing the crc32 checksum by combining the previous checksum, `OldCrc`, with the checksum of `Data`.

The following code:

```
X = crc32(Data1),
Y = crc32(X,Data2).
```

- would assign the same value to Y as this would:

```
Y = crc32([Data1,Data2]).
```

crc32_combine(FirstCrc, SecondCrc, SecondSize) -> int()

Types:

FirstCrc = SecondCrc = int()

SecondSize = int()

Combines two previously computed crc32 checksums. This computation requires the size of the data object for the second checksum to be known.

The following code:

```
Y = crc32(Data1),
Z = crc32(Y,Data2).
```

- would assign the same value to Z as this would:

```
X = crc32(Data1),
Y = crc32(Data2),
Z = crc32_combine(X,Y,iolist_size(Data2)).
```

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

Types:

Year = Month = Day = int()

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

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

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

decode_packet(Type,Bin,Options) -> {ok,Packet,Rest} | {more,Length} | {error,Reason}

Types:

Bin = binary()

Options = [Opt]

Packet = binary() | HttpPacket

Rest = binary()

Length = int() | undefined

Reason = term()

Type, Opt -- see below
HttpPacket = HttpRequest | HttpResponse | HttpHeader | http_eoh | HttpError
HttpRequest = {http_request, HttpMethod, HttpUri, HttpVersion}
HttpResponse = {http_response, HttpVersion, integer(), HttpString}
HttpHeader = {http_header, int(), HttpField, Reserved=term(), Value=HttpString}
HttpError = {http_error, HttpString}
HttpMethod = HttpMethodAtom | HttpString
HttpMethodAtom = 'OPTIONS' | 'GET' | 'HEAD' | 'POST' | 'PUT' | 'DELETE' | 'TRACE'
HttpUri = '*' | {absoluteURI, http|https, Host=HttpString, Port=int()|undefined, Path=HttpString} | {scheme, Scheme=HttpString, HttpString} | {abs_path, HttpString} | HttpString
HttpVersion = {Major=int(), Minor=int()}
HttpString = string() | binary()
HttpField = HttpFieldAtom | HttpString
HttpFieldAtom = 'Cache-Control' | 'Connection' | 'Date' | 'Pragma' | 'Transfer-Encoding' | 'Upgrade' | 'Via' | 'Accept' | 'Accept-Charset' | 'Accept-Encoding' | 'Accept-Language' | 'Authorization' | 'From' | 'Host' | 'If-Modified-Since' | 'If-Match' | 'If-None-Match' | 'If-Range' | 'If-Unmodified-Since' | 'Max-Forwards' | 'Proxy-Authorization' | 'Range' | 'Referer' | 'User-Agent' | 'Age' | 'Location' | 'Proxy-Authenticate' | 'Public' | 'Retry-After' | 'Server' | 'Vary' | 'Warning' | 'Www-Authenticate' | 'Allow' | 'Content-Base' | 'Content-Encoding' | 'Content-Language' | 'Content-Length' | 'Content-Location' | 'Content-Md5' | 'Content-Range' | 'Content-Type' | 'Etag' | 'Expires' | 'Last-Modified' | 'Accept-Ranges' | 'Set-Cookie' | 'Set-Cookie2' | 'X-Forwarded-For' | 'Cookie' | 'Keep-Alive' | 'Proxy-Connection'

Decodes the binary `Bin` according to the packet protocol specified by `Type`. Very similar to the packet handling done by sockets with the option {packet,Type}.

If an entire packet is contained in `Bin` it is returned together with the remainder of the binary as `{ok,Packet,Rest}`.

If `Bin` does not contain the entire packet, `{more,Length}` is returned. `Length` is either the expected *total size* of the packet or `undefined` if the expected packet size is not known. `decode_packet` can then be called again with more data added.

If the packet does not conform to the protocol format `{error,Reason}` is returned.

The following values of `Type` are valid:

`raw | 0`

No packet handling is done. Entire binary is returned unless it is empty.

`1 | 2 | 4`

Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of header can be one, two, or four bytes; the order of the bytes is big-endian. The header will be stripped off when the packet is returned.

`line`

A packet is a line terminated with newline. The newline character is included in the returned packet unless the line was truncated according to the option `line_length`.

`asn1 | cdr | sunrm | fcgi | tpkt`

The header is *not* stripped off.

The meanings of the packet types are as follows:

`asn1` - ASN.1 BER

`sunrm` - Sun's RPC encoding
`cdr` - CORBA (GIOP 1.1)
`fcgi` - Fast CGI
`tpkt` - TPKT format [RFC1006]

`http | httph | http_bin | httph_bin`

The Hypertext Transfer Protocol. The packets are returned with the format according to `HttpPacket` described above. A packet is either a request, a response, a header or an end of header mark. Invalid lines are returned as `HttpError`.

Recognized request methods and header fields are returned as atoms. Others are returned as strings.

The protocol type `http` should only be used for the first line when a `HttpRequest` or a `HttpResponse` is expected. The following calls should use `httph` to get `HttpHeader`'s until `http_eoh` is returned that marks the end of the headers and the beginning of any following message body.

The variants `http_bin` and `httph_bin` will return strings (`HttpString`) as binaries instead of lists.

The following options are available:

`{packet_size, int()}`

Sets the max allowed size of the packet body. If the packet header indicates that the length of the packet is longer than the max allowed length, the packet is considered invalid. Default is 0 which means no size limit.

`{line_length, int()}`

Applies only to line oriented protocols (`line`, `http`). Lines longer than this will be truncated.

```
> erlang:decode_packet(1,<<3,"abcd">>,[]).
{ok,<<"abc">>,<<"d">>}
> erlang:decode_packet(1,<<5,"abcd">>,[]).
{more,6}
```

`delete_module(Module) -> true | undefined`

Types:

Module = **atom()**

Makes the current code for `Module` become old code, and deletes all references for this module from the export table. Returns `undefined` if the module does not exist, otherwise `true`.

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

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

`erlang:demonitor(MonitorRef) -> true`

Types:

MonitorRef = **ref()**

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

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

> ### Note:
>
> Prior to OTP release R11B (erts version 5.5) `erlang:demonitor/1` behaved completely asynchronous, i.e., the monitor was active until the "demonitor signal" reached the monitored entity. This had one undesirable effect, though. You could never know when you were guaranteed *not* to receive a `DOWN` message due to the monitor.
>
> Current behavior can be viewed as two combined operations: asynchronously send a "demonitor signal" to the monitored entity and ignore any future results of the monitor.

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

`erlang:demonitor(MonitorRef, OptionList) -> true|false`

Types:

MonitorRef = ref()

OptionList = [Option]

Option = flush

Option = info

The returned value is `true` unless `info` is part of `OptionList`.

`erlang:demonitor(MonitorRef, [])` is equivalent to *erlang:demonitor(MonitorRef)*.

Currently the following `Option`s are valid:

`flush`

Remove (one) `{_, MonitorRef, _, _, _}` message, if there is one, from the callers message queue after monitoring has been stopped.

Calling `erlang:demonitor(MonitorRef, [flush])` is equivalent to the following, but more efficient:

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

`info`

The returned value is one of the following:

`true`

The monitor was found and removed. In this case no `'DOWN'` message due to this monitor have been nor will be placed in the message queue of the caller.

`false`

The monitor was not found and could not be removed. This probably because someone already has placed a `'DOWN'` message corresponding to this monitor in the callers message queue.

If the `info` option is combined with the `flush` option, `false` will be returned if a flush was needed; otherwise, `true`.

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

Failure: `badarg` if `OptionList` is not a list, or if `Option` is not a valid option, or the same failure as for *erlang:demonitor/1*

`disconnect_node(Node) -> bool() | ignored`

Types:

Node = atom()

Forces the disconnection of a node. This will appear to the node `Node` as if the local node has crashed. This BIF is mainly used in the Erlang network authentication protocols. Returns `true` if disconnection succeeds, otherwise `false`. If the local node is not alive, the function returns `ignored`.

`erlang:display(Term) -> true`

Types:

Term = term()

Prints a text representation of `Term` on the standard output.

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

`element(N, Tuple) -> term()`

Types:

N = 1..tuple_size(Tuple)

Tuple = tuple()

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

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

Allowed in guard tests.

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

Types:

Key = Val = term()

Returns the process dictionary and deletes it.

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

erase(Key) -> Val | undefined

Types:

Key = Val = term()

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

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

erlang:error(Reason)

Types:

Reason = term()

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

```
> catch erlang:error(foobar).
{'EXIT',{foobar,[{erl_eval,do_apply,5},
                 {erl_eval,expr,5},
                 {shell,exprs,6},
                 {shell,eval_exprs,6},
                 {shell,eval_loop,3}]}}
```

erlang:error(Reason, Args)

Types:

Reason = term()

Args = [term()]

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

exit(Reason)

Types:

Reason = term()

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

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

exit(Pid, Reason) -> true

Types:

Pid = pid()

Reason = term()

Sends an exit signal with exit reason `Reason` to the process `Pid`.

The following behavior apply if `Reason` is any term except `normal` or `kill`:

If `Pid` is not trapping exits, `Pid` itself will exit with exit reason `Reason`. If `Pid` is trapping exits, the exit signal is transformed into a message `{'EXIT', From, Reason}` and delivered to the message queue of `Pid`. `From` is the pid of the process which sent the exit signal. See also *process_flag/2*.

If `Reason` is the atom `normal`, `Pid` will not exit. If it is trapping exits, the exit signal is transformed into a message `{'EXIT', From, normal}` and delivered to its message queue.

If `Reason` is the atom `kill`, that is if `exit(Pid, kill)` is called, an untrappable exit signal is sent to `Pid` which will unconditionally exit with exit reason `killed`.

float(Number) -> float()

Types:

Number = number()

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

```
> float(55).
55.0
```

Allowed in guard tests.

> Note:
>
> Note that if used on the top-level in a guard, it will test whether the argument is a floating point number; for clarity, use *is_float/1* instead.
>
> When `float/1` is used in an expression in a guard, such as '`float(A) == 4.0`', it converts a number as described above.

`float_to_list(Float) -> string()`

Types:

Float = float()

Returns a string which corresponds to the text representation of `Float`.

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

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

Types:

Fun = fun()

Item, Info -- see below

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

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

There are two types of funs with slightly different semantics:

A fun created by `fun M:F/A` is called an *external* fun. Calling it will always call the function `F` with arity `A` in the latest code for module `M`. Note that module `M` does not even need to be loaded when the fun `fun M:F/A` is created.

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

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

`{type, Type}`

`Type` is either `local` or `external`.

`{module, Module}`

`Module` (an atom) is the module name.

If `Fun` is a local fun, `Module` is the module in which the fun is defined.

If `Fun` is an external fun, `Module` is the module that the fun refers to.

`{name, Name}`

`Name` (an atom) is a function name.

If `Fun` is a local fun, `Name` is the name of the local function that implements the fun. (This name was generated by the compiler, and is generally only of informational use. As it is a local function, it is not possible to call it directly.) If no code is currently loaded for the fun, `[]` will be returned instead of an atom.

If `Fun` is an external fun, `Name` is the name of the exported function that the fun refers to.

`{arity, Arity}`

`Arity` is the number of arguments that the fun should be called with.

`{env, Env}`

`Env` (a list) is the environment or free variables for the fun. (For external funs, the returned list is always empty.)

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

`{pid, Pid}`

`Pid` is the pid of the process that originally created the fun.

`{index, Index}`

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

`{new_index, Index}`

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

`{new_uniq, Uniq}`

`Uniq` (a binary) is a unique value for this fun.

`{uniq, Uniq}`

`Uniq` (an integer) is a unique value for this fun.

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

Types:

Fun = fun()

Item, Info -- see below

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

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

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

See *erlang:fun_info/1*.

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

Types:

Fun = fun()

Returns a string which corresponds to the text representation of `Fun`.

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

Types:

Module = Function = atom()

Arity = int()

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

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

`garbage_collect() -> true`

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

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

`garbage_collect(Pid) -> bool()`

Types:

Pid = pid()

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

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

Types:

Key = Val = term()

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

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

`get(Key) -> Val | undefined`

Types:

Key = Val = term()

Returns the value `Val` associated with `Key` in the process dictionary, or `undefined` if `Key` does not exist.

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

`erlang:get_cookie() -> Cookie | nocookie`

Types:

Cookie = atom()

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

`get_keys(Val) -> [Key]`

Types:

Val = Key = term()

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

```
> put(mary, {1, 2}),
put(had, {1, 2}),
```

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

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

Types:

Module = **Function** = **atom()**

Arity = **int()**

Args = **[term()]**

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

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

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

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

See also *erlang:error/1* and *erlang:error/2*.

group_leader() -> GroupLeader

Types:

GroupLeader = **pid()**

Returns the pid of the group leader for the process which evaluates the function.

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

group_leader(GroupLeader, Pid) -> true

Types:

GroupLeader = **Pid** = **pid()**

Sets the group leader of `Pid` to `GroupLeader`. Typically, this is used when a processes started from a certain shell should have another group leader than `init`.

See also *group_leader/0*.

halt()

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

```
> halt().
os_prompt%
```

halt(Status)

Types:

Status = int()>=0 | string()

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

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

`erlang:hash(Term, Range) -> Hash`

Returns a hash value for `Term` within the range `1..Range`. The allowed range is 1..2^27-1.

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

`hd(List) -> term()`

Types:

List = [term()]

Returns the head of `List`, that is, the first element.

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

Allowed in guard tests.

Failure: `badarg` if `List` is the empty list [].

`erlang:hibernate(Module, Function, Args)`

Types:

Module = Function = atom()

Args = [term()]

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

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

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

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

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

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

integer_to_list(Integer) -> string()

Types:

Integer = int()

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

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

erlang:integer_to_list(Integer, Base) -> string()

Types:

Integer = int()

Base = 2..36

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

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

iolist_to_binary(IoListOrBinary) -> binary()

Types:

IoListOrBinary = iolist() | binary()

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

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

iolist_size(Item) -> int()

Types:

Item = iolist() | binary()

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

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

is_alive() -> bool()

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

is_atom(Term) -> bool()

Types:

Term = **term()**

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

Allowed in guard tests.

is_binary(Term) -> bool()

Types:

Term = **term()**

Returns `true` if `Term` is a binary; otherwise returns `false`.

A binary always contains a complete number of bytes.

Allowed in guard tests.

is_bitstring(Term) -> bool()

Types:

Term = **term()**

Returns `true` if `Term` is a bitstring (including a binary); otherwise returns `false`.

Allowed in guard tests.

is_boolean(Term) -> bool()

Types:

Term = **term()**

Returns `true` if `Term` is either the atom `true` or the atom `false` (i.e. a boolean); otherwise returns `false`.

Allowed in guard tests.

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

Types:

Module = **Function** = **atom()**

Arity = **int()**

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

is_float(Term) -> bool()

Types:

Term = **term()**

Returns `true` if `Term` is a floating point number; otherwise returns `false`.

Allowed in guard tests.

`is_function(Term) -> bool()`

Types:

Term = **term()**

Returns `true` if `Term` is a fun; otherwise returns `false`.

Allowed in guard tests.

`is_function(Term, Arity) -> bool()`

Types:

Term = **term()**

Arity = **int()**

Returns `true` if `Term` is a fun that can be applied with `Arity` number of arguments; otherwise returns `false`.

Allowed in guard tests.

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

`is_integer(Term) -> bool()`

Types:

Term = **term()**

Returns `true` if `Term` is an integer; otherwise returns `false`.

Allowed in guard tests.

`is_list(Term) -> bool()`

Types:

Term = **term()**

Returns `true` if `Term` is a list with zero or more elements; otherwise returns `false`.

Allowed in guard tests.

`is_number(Term) -> bool()`

Types:

Term = **term()**

Returns `true` if `Term` is either an integer or a floating point number; otherwise returns `false`.

Allowed in guard tests.

is_pid(Term) -> bool()

Types:

Term = term()

Returns `true` if `Term` is a pid (process identifier); otherwise returns `false`.

Allowed in guard tests.

is_port(Term) -> bool()

Types:

Term = term()

Returns `true` if `Term` is a port identifier; otherwise returns `false`.

Allowed in guard tests.

is_process_alive(Pid) -> bool()

Types:

Pid = pid()

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

is_record(Term, RecordTag) -> bool()

Types:

Term = term()

RecordTag = atom()

Returns `true` if `Term` is a tuple and its first element is `RecordTag`. Otherwise, returns `false`.

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

Allowed in guard tests, if `RecordTag` is a literal atom.

is_record(Term, RecordTag, Size) -> bool()

Types:

Term = term()

RecordTag = atom()

Size = int()

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

Allowed in guard tests, provided that `RecordTag` is a literal atom and `Size` is a literal integer.

> Note:
>
> This BIF is documented for completeness. In most cases `is_record/2` should be used.

`is_reference(Term) -> bool()`

Types:

Term = term()

Returns `true` if `Term` is a reference; otherwise returns `false`.

Allowed in guard tests.

`is_tuple(Term) -> bool()`

Types:

Term = term()

Returns `true` if `Term` is a tuple; otherwise returns `false`.

Allowed in guard tests.

`length(List) -> int()`

Types:

List = [term()]

Returns the length of `List`.

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

Allowed in guard tests.

`link(Pid) -> true`

Types:

Pid = pid() | port()

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

If `Pid` does not exist, the behavior of the BIF depends on if the calling process is trapping exits or not (see *process_flag/2*):

- If the calling process is not trapping exits, and checking `Pid` is cheap -- that is, if `Pid` is local -- `link/1` fails with reason `noproc`.
- Otherwise, if the calling process is trapping exits, and/or `Pid` is remote, `link/1` returns `true`, but an exit signal with reason `noproc` is sent to the calling process.

`list_to_atom(String) -> atom()`

Types:

String = string()

Returns the atom whose text representation is `String`.

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

list_to_binary(IoList) -> binary()

Types:

IoList = iolist()

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

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

list_to_bitstring(BitstringList) -> bitstring()

Types:

BitstringList = [BitstringList | bitstring() | char()]

Returns a bitstring which is made from the integers and bitstrings in `BitstringList`. (The last tail in `BitstringList` is allowed to be a bitstring.)

```
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6,7:4,>>.
<<6>>
> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6,7:46>>
```

list_to_existing_atom(String) -> atom()

Types:

String = string()

Returns the atom whose text representation is `String`, but only if there already exists such atom.

Failure: `badarg` if there does not already exist an atom whose text representation is `String`.

list_to_float(String) -> float()

Types:

String = string()

Returns the float whose text representation is `String`.

```
> list_to_float("2.2017764e+0").
```

```
2.2017764
```

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

`list_to_integer(String) -> int()`

Types:

String = string()

Returns an integer whose text representation is `String`.

```
> list_to_integer("123").
123
```

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

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

Types:

String = string()

Base = 2..36

Returns an integer whose text representation in base `Base` is `String`.

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

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

`list_to_pid(String) -> pid()`

Types:

String = string()

Returns a pid whose text representation is `String`.

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

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

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

`list_to_tuple(List) -> tuple()`

Types:

List = [term()]

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

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

`load_module(Module, Binary) -> {module, Module} | {error, Reason}`

Types:

Module = atom()

Binary = binary()

Reason = badfile | not_purged | badfile

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

`badfile`

The object code in `Binary` has an incorrect format.

`not_purged`

`Binary` contains a module which cannot be loaded because old code for this module already exists.

`badfile`

The object code contains code for another module than `Module`

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

`erlang:load_nif(Path, LoadInfo) -> ok | {error, {Reason, Text}}`

Types:

Path = string()

LoadInfo = term()

Reason = load_failed | bad_lib | load | reload | upgrade | old_code

Text = string()

> Warning:
>
> This BIF is still an experimental feature. The interface may be changed in any way in future releases.
>
> In R13B03 the return value on failure was `{error,Reason,Text}`.

Loads and links a dynamic library containing native implemented functions (NIFs) for a module. `Path` is a file path to the sharable object/dynamic library file minus the OS-dependant file extension (.so for Unix and .ddl for Windows). See *erl_nif* on how to implement a NIF library.

`LoadInfo` can be any term. It will be passed on to the library as part of the initialization. A good practice is to include a module version number to support future code upgrade scenarios.

The call to `load_nif/2` must be made *directly* from the Erlang code of the module that the NIF library belongs to.

It returns either `ok`, or `{error,{Reason,Text}}` if loading fails. `Reason` is one of the atoms below, while `Text` is a human readable string that may give some more information about the failure.

`load_failed`

The OS failed to load the NIF library.

`bad_lib`

The library did not fulfil the requirements as a NIF library of the calling module.

`load | reload | upgrade`

The corresponding library callback was not successful.

`old_code`

The call to `load_nif/2` was made from the old code of a module that has been upgraded. This is not allowed.

erlang:loaded() -> [Module]

Types:

Module = atom()

Returns a list of all loaded Erlang modules (current and/or old code), including preloaded modules.

See also *code(3)*.

erlang:localtime() -> {Date, Time}

Types:

Date = {Year, Month, Day}

Time = {Hour, Minute, Second}

Year = Month = Day = Hour = Minute = Second = int()

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

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

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

erlang:localtime_to_universaltime({Date1, Time1}) -> {Date2, Time2}

Types:

Date1 = Date2 = {Year, Month, Day}

Time1 = Time2 = {Hour, Minute, Second}

Year = Month = Day = Hour = Minute = Second = int()

Converts local date and time to Universal Time Coordinated (UTC), if this is supported by the underlying OS. Otherwise, no conversion is done and `{Date1, Time1}` is returned.

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

```
{{1996,11,6},{13,45,17}}
```

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

erlang:localtime_to_universaltime({Date1, Time1}, IsDst) -> {Date2, Time2}

Types:

Date1 = Date2 = {Year, Month, Day}

Time1 = Time2 = {Hour, Minute, Second}

Year = Month = Day = Hour = Minute = Second = int()

IsDst = true | false | undefined

Converts local date and time to Universal Time Coordinated (UTC) just like `erlang:localtime_to_universaltime/1`, but the caller decides if daylight saving time is active or not.

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

```
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, true).
{{1996,11,6},{12,45,17}}
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, false).
{{1996,11,6},{13,45,17}}
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, undefined).
{{1996,11,6},{13,45,17}}
```

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

make_ref() -> ref()

Returns an almost unique reference.

The returned reference will re-occur after approximately 2^82 calls; therefore it is unique enough for practical purposes.

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

erlang:make_tuple(Arity, InitialValue) -> tuple()

Types:

Arity = int()

InitialValue = term()

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

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

erlang:make_tuple(Arity, Default, InitList) -> tuple()

Types:

Arity = int()

Default = term()

InitList = [{Position,term()}]

Position = integer()

`erlang:make_tuple` first creates a tuple of size `Arity` where each element has the value `Default`. It then fills in values from `InitList`. Each list element in `InitList` must be a two-tuple where the first element is a position in the newly created tuple and the second element is any term. If a position occurs more than once in the list, the term corresponding to last occurrence will be used.

```
> erlang:make_tuple(5, [], [{2,ignored},{5,zz},{2,aa}]).
{{[],aa,[],[],zz}
```

`erlang:max(Term1, Term2) -> Maximum`

Types:

Term1 = Term2 = Maximum = term()

Return the largest of `Term1` and `Term2`; if the terms compares equal, `Term1` will be returned.

`erlang:md5(Data) -> Digest`

Types:

Data = iodata()

Digest = binary()

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

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

> Warning:
>
> The MD5 Message Digest Algorithm is *not* considered safe for code-signing or software integrity purposes.

`erlang:md5_final(Context) -> Digest`

Types:

Context = Digest = binary()

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

`erlang:md5_init() -> Context`

Types:

Context = binary()

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

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

Types:

Data = iodata()

Context = NewContext = binary()

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

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

Types:

Type, Size -- see below

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

`total`

The total amount of memory currently allocated, which is the same as the sum of memory size for `processes` and `system`.

`processes`

The total amount of memory currently allocated by the Erlang processes.

`processes_used`

The total amount of memory currently used by the Erlang processes.

This memory is part of the memory presented as `processes` memory.

`system`

The total amount of memory currently allocated by the emulator that is not directly related to any Erlang process.

Memory presented as `processes` is not included in this memory.

`atom`

The total amount of memory currently allocated for atoms.

This memory is part of the memory presented as `system` memory.

`atom_used`

The total amount of memory currently used for atoms.

This memory is part of the memory presented as `atom` memory.

`binary`

The total amount of memory currently allocated for binaries.

This memory is part of the memory presented as `system` memory.

`code`

The total amount of memory currently allocated for Erlang code.

This memory is part of the memory presented as `system` memory.

`ets`

The total amount of memory currently allocated for ets tables.

This memory is part of the memory presented as `system` memory.

`maximum`

The maximum total amount of memory allocated since the emulator was started.

This tuple is only present when the emulator is run with instrumentation.

For information on how to run the emulator with instrumentation see *instrument(3)* and/or *erl(1)*.

> ### Note:
>
> The `system` value is not complete. Some allocated memory that should be part of the `system` value are not.
>
> When the emulator is run with instrumentation, the `system` value is more accurate, but memory directly allocated by `malloc` (and friends) are still not part of the `system` value. Direct calls to `malloc` are only done from OS specific runtime libraries and perhaps from user implemented Erlang drivers that do not use the memory allocation functions in the driver interface.
>
> Since the `total` value is the sum of `processes` and `system` the error in `system` will propagate to the `total` value.
>
> The different amounts of memory that are summed are *not* gathered atomically which also introduce an error in the result.

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

```
total = processes + system
processes = processes_used + ProcessesNotUsed
system = atom + binary + code + ets + OtherSystem
atom = atom_used + AtomNotUsed

RealTotal = processes + RealSystem
RealSystem = system + MissedSystem
```

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

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

> ### Note:
>
> Since erts version 5.6.4 `erlang:memory/0` requires that all *erts_alloc(3)* allocators are enabled (default behaviour).

Failure:

`notsup`
: If an *erts_alloc(3)* allocator has been disabled.

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

Types:

Type, Size -- see below

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

> Note:
>
> Since erts version 5.6.4 `erlang:memory/1` requires that all *erts_alloc(3)* allocators are enabled (default behaviour).

Failures:

`badarg`
If `Type` is not one of the memory types listed in the documentation of *erlang:memory/0*.

`badarg`
If `maximum` is passed as `Type` and the emulator is not run in instrumented mode.

`notsup`
If an *erts_alloc(3)* allocator has been disabled.

See also *erlang:memory/0*.

erlang:min(Term1, Term2) -> Minimum

Types:

Term1 = Term2 = Minimum = term()

Return the smallest of `Term1` and `Term2`; if the terms compare equal, `Term1` will be returned.

module_loaded(Module) -> bool()

Types:

Module = atom()

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

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

erlang:monitor(Type, Item) -> MonitorRef

Types:

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

The calling process starts monitoring `Item` which is an object of type `Type`.

Currently only processes can be monitored, i.e. the only allowed `Type` is `process`, but other types may be allowed in the future.

`Item` can be:

`pid()`

The pid of the process to monitor.

`{RegName, Node}`

A tuple consisting of a registered name of a process and a node name. The process residing on the node `Node` with the registered name `RegName` will be monitored.

`RegName`

The process locally registered as `RegName` will be monitored.

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

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

```
{'DOWN', MonitorRef, Type, Object, Info}
```

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

`Object`

A reference to the monitored object:

- the pid of the monitored process, if `Item` was specified as a pid.
- `{RegName, Node}`, if `Item` was specified as `{RegName, Node}`.
- `{RegName, Node}`, if `Item` was specified as `RegName`. `Node` will in this case be the name of the local node (`node()`).

`Info`

Either the exit reason of the process, `noproc` (non-existing process), or `noconnection` (no connection to `Node`).

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

The monitoring is turned off either when the `'DOWN'` message is sent, or when *erlang:demonitor/1* is called.

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

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

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

`monitor_node(Node, Flag) -> true`

Types:

Node = node()

Flag = bool()

Monitors the status of the node `Node`. If `Flag` is `true`, monitoring is turned on; if `Flag` is `false`, monitoring is turned off.

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

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

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

Failure: `badarg`if the local node is not alive.

`erlang:monitor_node(Node, Flag, Options) -> true`

Types:

Node = node()

Flag = bool()

Options = [Option]

Option = allow_passive_connect

Behaves as `monitor_node/2` except that it allows an extra option to be given, namely `allow_passive_connect`. The option allows the BIF to wait the normal net connection timeout for the *monitored node* to connect itself, even if it cannot be actively connected from this node (i.e. it is blocked). The state where this might be useful can only be achieved by using the kernel option `dist_auto_connect once`. If that kernel option is not used, the `allow_passive_connect` option has no effect.

> Note:
>
> The `allow_passive_connect` option is used internally and is seldom needed in applications where the network topology and the kernel options in effect is known in advance.

Failure: `badarg` if the local node is not alive or the option list is malformed.

node() -> Node

Types:

Node = **node()**

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

Allowed in guard tests.

node(Arg) -> Node

Types:

Arg = **pid()** | **port()** | **ref()**

Node = **node()**

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

Allowed in guard tests.

nodes() -> Nodes

Types:

Nodes = **[node()]**

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

nodes(Arg | [Arg]) -> Nodes

Types:

Arg = **visible** | **hidden** | **connected** | **this** | **known**

Nodes = **[node()]**

Returns a list of nodes according to argument given. The result returned when the argument is a list, is the list of nodes satisfying the disjunction(s) of the list elements.

`Arg` can be any of the following:

`visible`

Nodes connected to this node through normal connections.

`hidden`

Nodes connected to this node through hidden connections.

`connected`

All nodes connected to this node.

`this`

This node.

`known`

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

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

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

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

Types:

MegaSecs = Secs = MicroSecs = int()

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

`open_port(PortName, PortSettings) -> port()`

Types:

PortName = {spawn, Command} | {spawn_driver, Command} | {spawn_executable, Command} | {fd, In, Out}

Command = string()

In = Out = int()

PortSettings = [Opt]

Opt = {packet, N} | stream | {line, L} | {cd, Dir} | {env, Env} | {args, [string()]} | {arg0, string()} | exit_status | use_stdio | nouse_stdio | stderr_to_stdout | in | out | binary | eof

N = 1 | 2 | 4

L = int()

Dir = string()

Env = [{Name, Val}]

Name = string()

Val = string() | false

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

`{spawn, Command}`

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

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

For external programs, the `PATH` is searched (or an equivalent method is used to find programs, depending on operating system). This is done by invoking the shell och certain platforms. The first space separated token of the command will be considered as the name of the executable (or driver). This (among other things) makes this option unsuitable for running programs having spaces in file or directory names. Use {spawn_executable, Command} instead if spaces in executable file names is desired.

`{spawn_driver, Command}`

Works like `{spawn, Command}`, but demands the first (space separated) token of the command to be the name of a loaded driver. If no driver with that name is loaded, a `badarg` error is raised.

`{spawn_executable, Command}`

Works like `{spawn, Command}`, but only runs external executables. The `Command` in it's whole is used as the name of the executable, including any spaces. If arguments are to be passed, the `args` and `arg0` `PortSettings` can be used.

The shell is not usually invoked to start the program, it's executed directly. Neither is the `PATH` (or equivalent) searched. To find a program in the PATH to execute, use *os:find_executable/1*.

Only if a shell script or `.bat` file is executed, the appropriate command interpreter will implicitly be invoked, but there will still be no command argument expansion or implicit PATH search.

If the `Command` cannot be run, an error exception, with the posix error code as the reason, is raised. The error reason may differ between operating systems. Typically the error `enoent` is raised when one tries to run a program that is not found and `eaccess` is raised when the given file is not executable.

`{fd, In, Out}`

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

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

`{packet, N}`

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

`stream`

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

`{line, L}`

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

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

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

`{cd, Dir}`

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

`{env, Env}`

This is only valid for `{spawn, Command}` and `{spawn_executable, Command}`. The environment of the started process is extended using the environment specifications in `Env`.

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

`{args, [ string() ]}`

This option is only valid for `{spawn_executable, Command}` and specifies arguments to the executable. Each argument is given as a separate string and (on Unix) eventually ends up as one element each in the argument vector. On other platforms, similar behavior is mimicked.

The arguments are not expanded by the shell prior to being supplied to the executable, most notably this means that file wildcard expansion will not happen. Use *filelib:wildcard/1* to expand wildcards for the arguments. Note that even if the program is a Unix shell script, meaning that the shell will ultimately be invoked, wildcard expansion will not happen and the script will be provided with the untouched arguments. On Windows®, wildcard expansion is always up to the program itself, why this isn't an issue.

Note also that the actual executable name (a.k.a. `argv[0]`) should not be given in this list. The proper executable name will automatically be used as argv[0] where applicable.

If one, for any reason, wants to explicitly set the program name in the argument vector, the `arg0` option can be used.

`{arg0, string()}`

This option is only valid for `{spawn_executable, Command}` and explicitly specifies the program name argument when running an executable. This might in some circumstances, on some operating systems, be desirable. How the program responds to this is highly system dependent and no specific effect is guaranteed.

`exit_status`

This is only valid for `{spawn, Command}` where `Command` refers to an external program, and for `{spawn_executable, Command}`.

When the external process connected to the port exits, a message of the form `{Port, {exit_status,Status}}` is sent to the connected process, where `Status` is the exit status of the external process. If the program aborts, on Unix the same convention is used as the shells do (i.e., 128+signal).

If the `eof` option has been given as well, the `eof` message and the `exit_status` message appear in an unspecified order.

If the port program closes its stdout without exiting, the `exit_status` option will not work.

`use_stdio`

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

`nouse_stdio`

The opposite of `use_stdio`. Uses file descriptors 3 and 4 for communication with Erlang.

`stderr_to_stdout`

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

`overlapped_io`

Affects ports to external programs on Windows® only. The standard input and standard output handles of the port program will, if this option is supplied, be opened with the flag FILE_FLAG_OVERLAPPED, so that the port program can (and has to) do overlapped I/O on it's standard handles. This is not normally the case for simple port programs, but an option of value for the experienced Windows programmer. *On all other platforms, this option is silently discarded.*

`in`

The port can only be used for input.

`out`

The port can only be used for output.

`binary`

All IO from the port are binary data objects as opposed to lists of bytes.

`eof`

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

`hide`

When running on Windows, suppress creation of a new console window when spawning the port program. (This option has no effect on other platforms.)

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

Failure: If the port cannot be opened, the exit reason is `badarg`, `system_limit`, or the Posix error code which most closely describes the error, or `einval` if no Posix code is appropriate:

`badarg`

Bad input arguments to `open_port`.

`system_limit`

All available ports in the Erlang emulator are in use.

`enomem`

There was not enough memory to create the port.

`eagain`

There are no more available operating system processes.

`enametoolong`

The external command given was too long.

`emfile`

There are no more available file descriptors (for the operating system process that the Erlang emulator runs in).

`enfile`

The file table is full (for the entire operating system).

`eacces`

The `Command` given in `{spawn_executable, Command}` does not point out an executable file.

`enoent`

The `Command` given in `{spawn_executable, Command}` does not point out an existing file.

During use of a port opened using `{spawn, Name}`, `{spawn_driver, Name}` or `{spawn_executable, Name}`, errors arising when sending messages to it are reported to the owning process using signals of the form `{'EXIT', Port, PosixCode}`. See `file(3)` for possible values of `PosixCode`.

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

`erlang:phash(Term, Range) -> Hash`

Types:

Term = term()

Range = 1..2^32

Hash = 1..Range

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

This BIF could be used instead of the old deprecated `erlang:hash/2` BIF, as it calculates better hashes for all data-types, but consider using `phash2/1,2` instead.

`erlang:phash2(Term [, Range]) -> Hash`

Types:

Term = term()

Range = 1..2^32

Hash = 0..Range-1

Portable hash function that will give the same hash for the same Erlang term regardless of machine architecture and ERTS version (the BIF was introduced in ERTS 5.2). Range can be between 1 and 2^32, the function returns a hash value for `Term` within the range `0..Range-1`. When called without the `Range` argument, a value in the range `0..2^27-1` is returned.

This BIF should always be used for hashing terms. It distributes small integers better than `phash/2`, and it is faster for bignums and binaries.

Note that the range `0..Range-1` is different from the range of `phash/2` (`1..Range`).

`pid_to_list(Pid) -> string()`

Types:

Pid = pid()

Returns a string which corresponds to the text representation of `Pid`.

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

`port_close(Port) -> true`

Types:

Port = port() | atom()

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

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

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

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

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

port_command(Port, Data) -> true

Types:

Port = port() | atom()

Data = iodata()

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

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

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

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

If the port is busy, the calling process will be suspended until the port is not busy anymore.

Failures:

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

`badarg`
: If `Data` is not a valid io list.

erlang:port_command(Port, Data, OptionList) -> true|false

Types:

Port = port() | atom()

Data = iodata()

OptionList = [Option]

Option = force

Option = nosuspend

Sends data to a port. `port_command(Port, Data, [])` equals `port_command(Port, Data)`.

If the port command is aborted `false` is returned; otherwise, `true` is returned.

If the port is busy, the calling process will be suspended until the port is not busy anymore.

Currently the following `Option`s are valid:

`force`
: The calling process will not be suspended if the port is busy; instead, the port command is forced through. The call will fail with a `notsup` exception if the driver of the port does not support this. For more information see the *ERL_DRV_FLAG_SOFT_BUSY* driver flag.

`nosuspend`
: The calling process will not be suspended if the port is busy; instead, the port command is aborted and `false` is returned.

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

> Note:
>
> `erlang:port_command/3` is currently not auto imported, but it is planned to be auto imported in OTP R14.

Failures:

`badarg`
If `Port` is not an open port or the registered name of an open port.

`badarg`
If `Data` is not a valid io list.

`badarg`
If `OptionList` is not a valid option list.

`notsup`
If the `force` option has been passed, but the driver of the port does not allow forcing through a busy port.

`port_connect(Port, Pid) -> true`

Types:

Port = port() | atom()

Pid = pid()

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

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

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

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

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

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

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

`port_control(Port, Operation, Data) -> Res`

Types:

Port = port() | atom()

Operation = int()

Data = Res = iodata()

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

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

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

`erlang:port_call(Port, Operation, Data) -> term()`

Types:

Port = port() | atom()

Operation = int()

Data = term()

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

`Port` is a port identifier, referring to a driver.

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

`Data` is any Erlang term. This data is converted to binary term format and sent to the port.

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

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

`erlang:port_info(Port) -> [{Item, Info}] | undefined`

Types:

Port = port() | atom()

Item, Info -- see below

Returns a list containing tuples with information about the `Port`, or `undefined` if the port is not open. The order of the tuples is not defined, nor are all the tuples mandatory.

`{registered_name, RegName}`

`RegName` (an atom) is the registered name of the port. If the port has no registered name, this tuple is not present in the list.

`{id, Index}`

`Index` (an integer) is the internal index of the port. This index may be used to separate ports.

`{connected, Pid}`

`Pid` is the process connected to the port.

`{links, Pids}`

`Pids` is a list of pids to which processes the port is linked.

`{name, String}`

`String` is the command name set by `open_port`.

`{input, Bytes}`

`Bytes` is the total number of bytes read from the port.

`{output, Bytes}`

`Bytes` is the total number of bytes written to the port.

Failure: `badarg` if `Port` is not a local port.

`erlang:port_info(Port, Item) -> {Item, Info} | undefined | []`

Types:

Port = port() | atom()

Item, Info -- see below

Returns information about `Port` as specified by `Item`, or `undefined` if the port is not open. Also, if `Item == registered_name` and the port has no registered name, [] is returned.

For valid values of `Item`, and corresponding values of `Info`, see *erlang:port_info/1*.

Failure: `badarg` if `Port` is not a local port.

`erlang:port_to_list(Port) -> string()`

Types:

Port = port()

Returns a string which corresponds to the text representation of the port identifier `Port`.

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

`erlang:ports() -> [port()]`

Returns a list of all ports on the local node.

`pre_loaded() -> [Module]`

Types:

Module = atom()

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

`erlang:process_display(Pid, Type) -> void()`

Types:

Pid = pid()

Type = backtrace

Writes information about the local process `Pid` on standard error. The currently allowed value for the atom `Type` is `backtrace`, which shows the contents of the call stack, including information about the call chain, with the current function printed first. The format of the output is not further defined.

`process_flag(Flag, Value) -> OldValue`

Types:

Flag, Value, OldValue -- see below

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

`process_flag(trap_exit, Boolean)`

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

See also *exit/2*.

`process_flag(error_handler, Module)`

This is used by a process to redefine the error handler for undefined function calls and undefined registered processes. Inexperienced users should not use this flag since code auto-loading is dependent on the correct operation of the error handling module.

`process_flag(min_heap_size, MinHeapSize)`

This changes the minimum heap size for the calling process.

`process_flag(min_bin_vheap_size, MinBinVHeapSize)`

This changes the minimum binary virtual heap size for the calling process.

`process_flag(priority, Level)`

This sets the process priority. `Level` is an atom. There are currently four priority levels: `low`, `normal`, `high`, and `max`. The default priority level is `normal`. *NOTE*: The `max` priority level is reserved for internal use in the Erlang runtime system, and should *not* be used by others.

Internally in each priority level processes are scheduled in a round robin fashion.

Execution of processes on priority `normal` and priority `low` will be interleaved. Processes on priority `low` will be selected for execution less frequently than processes on priority `normal`.

When there are runnable processes on priority `high` no processes on priority `low`, or `normal` will be selected for execution. Note, however, that this does *not* mean that no processes on priority `low`, or `normal` will be able to run when there are processes on priority `high` running. On the runtime system with SMP support there might be more processes running in parallel than processes on priority `high`, i.e., a `low`, and a `high` priority process might execute at the same time.

When there are runnable processes on priority `max` no processes on priority `low`, `normal`, or `high` will be selected for execution. As with the `high` priority, processes on lower priorities might execute in parallel with processes on priority `max`.

Scheduling is preemptive. Regardless of priority, a process is preempted when it has consumed more than a certain amount of reductions since the last time it was selected for execution.

NOTE: You should not depend on the scheduling to remain exactly as it is today. Scheduling, at least on the runtime system with SMP support, is very likely to be modified in the future in order to better utilize available processor cores.

There is currently *no* automatic mechanism for avoiding priority inversion, such as priority inheritance, or priority ceilings. When using priorities you have to take this into account and handle such scenarios by yourself.

Making calls from a `high` priority process into code that you don't have control over may cause the `high` priority process to wait for a processes with lower priority, i.e., effectively decreasing the priority of the `high` priority process during the call. Even if this isn't the case with one version of the code that you don't have under your control, it might be the case in a future version of it. This might, for example, happen if a `high` priority process triggers code loading, since the code server runs on priority `normal`.

Other priorities than `normal` are normally not needed. When other priorities are used, they need to be used with care, especially the `high` priority *must* be used with care. A process on `high` priority should only perform work for short periods of time. Busy looping for long periods of time in a `high` priority process will most likely cause problems, since there are important servers in OTP running on priority `normal`.

`process_flag(save_calls, N)`

When there are runnable processes on priority `max` no processes on priority `low`, `normal`, or `high` will be selected for execution. As with the `high` priority, processes on lower priorities might execute in parallel with processes on priority `max`.

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

`process_flag(sensitive, Boolean)`

Set or clear the `sensitive` flag for the current process. When a process has been marked as sensitive by calling `process_flag(sensitive, true)`, features in the run-time system that can be used for examining the data and/or inner working of the process are silently disabled.

Features that are disabled include (but are not limited to) the following:

Tracing: Trace flags can still be set for the process, but no trace messages of any kind will be generated. (If the `sensitive` flag is turned off, trace messages will again be generated if there are any trace flags set.)

Sequential tracing: The sequential trace token will be propagated as usual, but no sequential trace messages will be generated.

`process_info/1,2` cannot be used to read out the message queue or the process dictionary (both will be returned as empty lists).

Stack back-traces cannot be displayed for the process.

In crash dumps, the stack, messages, and the process dictionary will be omitted.

If `{save_calls,N}` has been set for the process, no function calls will be saved to the call saving list. (The call saving list will not be cleared; furthermore, send, receive, and timeout events will still be added to the list.)

`process_flag(Pid, Flag, Value) -> OldValue`

Types:

Pid = pid()

Flag, Value, OldValue -- see below

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

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

`process_info(Pid) -> InfoResult`

Types:

Pid = pid()

Item = atom()

Info = term()

InfoTuple = {Item, Info}

InfoTupleList = [InfoTuple]

InfoResult = InfoTupleList | undefined

Returns a list containing `InfoTuple`s with miscellaneous information about the process identified by `Pid`, or `undefined` if the process is not alive.

The order of the `InfoTuple`s is not defined, nor are all the `InfoTuple`s mandatory. The `InfoTuple`s part of the result may be changed without prior notice. Currently `InfoTuple`s with the following `Item`s are part of the result: `current_function`, `initial_call`, `status`, `message_queue_len`, `messages`, `links`, `dictionary`, `trap_exit`, `error_handler`, `priority`, `group_leader`, `total_heap_size`, `heap_size`, `stack_size`, `reductions`, and `garbage_collection`. If the process identified by `Pid` has a registered name also an `InfoTuple` with `Item == registered_name` will appear.

See *process_info/2* for information about specific `InfoTuple`s.

> ### Warning:
>
> This BIF is intended for *debugging only*, use *process_info/2* for all other purposes.

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

`process_info(Pid, ItemSpec) -> InfoResult`

Types:

Pid = pid()

Item = atom()

Info = term()

ItemList = [Item]

ItemSpec = Item | ItemList

InfoTuple = {Item, Info}

InfoTupleList = [InfoTuple]

InfoResult = InfoTuple | InfoTupleList | undefined | []

Returns information about the process identified by `Pid` as specified by the `ItemSpec`, or `undefined` if the process is not alive.

If the process is alive and `ItemSpec` is a single `Item`, the returned value is the corresponding `InfoTuple` unless `ItemSpec == registered_name` and the process has no registered name. In this case `[]` is returned. This strange behavior is due to historical reasons, and is kept for backward compatibility.

If `ItemSpec` is an `ItemList`, the result is an `InfoTupleList`. The `InfoTuple`s in the `InfoTupleList` will appear with the corresponding `Item`s in the same order as the `Item`s appeared in the `ItemList`. Valid `Item`s may appear multiple times in the `ItemList`.

> Note:
>
> If `registered_name` is part of an `ItemList` and the process has no name registered a `{registered_name, []}` `InfoTuple` *will* appear in the resulting `InfoTupleList`. This behavior is different than when `ItemSpec == registered_name`, and than when `process_info/1` is used.

Currently the following `InfoTuple`s with corresponding `Item`s are valid:

`{backtrace, Bin}`

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

`{binary, BinInfo}`

`BinInfo` is a list containing miscellaneous information about binaries currently being referred to by this process. This `InfoTuple` may be changed or removed without prior notice.

`{catchlevel, CatchLevel}`

`CatchLevel` is the number of currently active catches in this process. This `InfoTuple` may be changed or removed without prior notice.

`{current_function, {Module, Function, Args}}`

`Module`, `Function`, `Args` is the current function call of the process.

`{dictionary, Dictionary}`

`Dictionary` is the dictionary of the process.

`{error_handler, Module}`

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

`{garbage_collection, GCInfo}`

`GCInfo` is a list which contains miscellaneous information about garbage collection for this process. The content of `GCInfo` may be changed without prior notice.

`{group_leader, GroupLeader}`

`GroupLeader` is group leader for the IO of the process.

`{heap_size, Size}`

`Size` is the size in words of youngest heap generation of the process. This generation currently include the stack of the process. This information is highly implementation dependent, and may change if the implementation change.

`{initial_call, {Module, Function, Arity}}`

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

`{links, Pids}`

`Pids` is a list of pids, with processes to which the process has a link.

`{last_calls, false|Calls}`

The value is `false` if call saving is not active for the process (see *process_flag/3*). If call saving is active, a list is returned, in which the last element is the most recent called.

`{memory, Size}`

`Size` is the size in bytes of the process. This includes call stack, heap and internal structures.

`{message_binary, BinInfo}`

`BinInfo` is a list containing miscellaneous information about binaries currently being referred to by the message area. This `InfoTuple` is only valid on an emulator using the hybrid heap type. This `InfoTuple` may be changed or removed without prior notice.

`{message_queue_len, MessageQueueLen}`

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

`{messages, MessageQueue}`

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

`{min_heap_size, MinHeapSize}`

`MinHeapSize` is the minimum heap size for the process.

`{min_bin_vheap_size, MinBinVHeapSize}`

`MinBinVHeapSize` is the minimum binary virtual heap size for the process.

`{monitored_by, Pids}`

A list of pids that are monitoring the process (with `erlang:monitor/2`).

`{monitors, Monitors}`

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

`{priority, Level}`

`Level` is the current priority level for the process. For more information on priorities see *process_flag(priority, Level)*.

`{reductions, Number}`

`Number` is the number of reductions executed by the process.

`{registered_name, Atom}`

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

`{sequential_trace_token, [] | SequentialTraceToken}`

`SequentialTraceToken` the sequential trace token for the process. This `InfoTuple` may be changed or removed without prior notice.

`{stack_size, Size}`

`Size` is the stack size of the process in words.

`{status, Status}`

`Status` is the status of the process. `Status` is `waiting` (waiting for a message), `running`, `runnable` (ready to run, but another process is running), or `suspended` (suspended on a "busy" port or by the `erlang:suspend_process/[1,2]` BIF).

`{suspending, SuspendeeList}`

`SuspendeeList` is a list of `{Suspendee, ActiveSuspendCount, OutstandingSuspendCount}` tuples. `Suspendee` is the pid of a process that have been or is to be suspended by the process identified by `Pid` via the *erlang:suspend_process/2* BIF, or the *erlang:suspend_process/1* BIF. `ActiveSuspendCount` is the number of times the `Suspendee` has been

suspended by `Pid`. `OutstandingSuspendCount` is the number of not yet completed suspend requests sent by `Pid`. That is, if `ActiveSuspendCount /= 0`, `Suspendee` is currently in the suspended state, and if `OutstandingSuspendCount /= 0` the `asynchronous` option of `erlang:suspend_process/2` has been used and the suspendee has not yet been suspended by `Pid`. Note that the `ActiveSuspendCount` and `OutstandingSuspendCount` are not the total suspend count on `Suspendee`, only the parts contributed by `Pid`.

`{total_heap_size, Size}`

`Size` is the total size in words of all heap fragments of the process. This currently include the stack of the process.

`{trace, InternalTraceFlags}`

`InternalTraceFlags` is an integer representing internal trace flag for this process. This `InfoTuple` may be changed or removed without prior notice.

`{trap_exit, Boolean}`

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

Note however, that not all implementations support every one of the above `Items`.

Failure: `badarg` if `Pid` is not a local process, or if `Item` is not a valid `Item`.

`processes() -> [pid()]`

Returns a list of process identifiers corresponding to all the processes currently existing on the local node.

Note that a process that is exiting, exists but is not alive, i.e., `is_process_alive/1` will return `false` for a process that is exiting, but its process identifier will be part of the result returned from `processes/0`.

```
> processes().
[<0.0.0>,<0.2.0>,<0.4.0>,<0.5.0>,<0.7.0>,<0.8.0>]
```

`purge_module(Module) -> void()`

Types:

Module = atom()

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

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

Failure: `badarg` if there is no old code for `Module`.

`put(Key, Val) -> OldVal | undefined`

Types:

Key = Val = OldVal = term()

Adds a new `Key` to the process dictionary, associated with the value `Val`, and returns `undefined`. If `Key` already exists, the old value is deleted and replaced by `Val` and the function returns the old value.

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

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

`erlang:raise(Class, Reason, Stacktrace)`

Types:

Class = error | exit | throw

Reason = term()

Stacktrace = [{Module, Function, Arity | Args} | {Fun, Args}]

Module = Function = atom()

Arity = int()

Args = [term()]

Fun = [fun()]

Stops the execution of the calling process with an exception of given class, reason and call stack backtrace (*stacktrace*).

> **Warning:**
>
> This BIF is intended for debugging and for use in the Erlang operating system. In general, it should be avoided in applications, unless you know very well what you are doing.

`Class` is one of `error`, `exit` or `throw`, so if it were not for the stacktrace `erlang:raise(Class, Reason, Stacktrace)` is equivalent to `erlang:Class(Reason)`. `Reason` is any term and `Stacktrace` is a list as returned from `get_stacktrace()`, that is a list of 3-tuples `{Module, Function, Arity | Args}` where `Module` and `Function` are atoms and the third element is an integer arity or an argument list. The stacktrace may also contain `{Fun, Args}` tuples where `Fun` is a local fun and `Args` is an argument list.

The stacktrace is used as the exception stacktrace for the calling process; it will be truncated to the current maximum stacktrace depth.

Because evaluating this function causes the process to terminate, it has no return value - unless the arguments are invalid, in which case the function *returns the error reason*, that is `badarg`. If you want to be really sure not to return you can call `erlang:error(erlang:raise(Class, Reason, Stacktrace))` and hope to distinguish exceptions later.

`erlang:read_timer(TimerRef) -> int() | false`

Types:

TimerRef = ref()

`TimerRef` is a timer reference returned by *erlang:send_after/3* or *erlang:start_timer/3*. If the timer is active, the function returns the time in milliseconds left until the timer will expire, otherwise `false` (which means that `TimerRef` was never a timer, that it has been cancelled, or that it has already delivered its message).

See also *erlang:send_after/3*, *erlang:start_timer/3*, and *erlang:cancel_timer/1*.

`erlang:ref_to_list(Ref) -> string()`

Types:

Ref = ref()

Returns a string which corresponds to the text representation of `Ref`.

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

`register(RegName, Pid | Port) -> true`

Types:

RegName = atom()

Pid = pid()

Port = port()

Associates the name `RegName` with a pid or a port identifier. `RegName`, which must be an atom, can be used instead of the pid / port identifier in the send operator (`RegName ! Message`).

```
> register(db, Pid).
true
```

Failure: `badarg` if `Pid` is not an existing, local process or port, if `RegName` is already in use, if the process or port is already registered (already has a name), or if `RegName` is the atom `undefined`.

`registered() -> [RegName]`

Types:

RegName = atom()

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

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

`erlang:resume_process(Suspendee) -> true`

Types:

Suspendee = pid()

Decreases the suspend count on the process identified by `Suspendee`. `Suspendee` should previously have been suspended via *erlang:suspend_process/2*, or *erlang:suspend_process/1* by the process calling `erlang:resume_process(Suspendee)`. When the suspend count on `Suspendee` reach zero, `Suspendee` will be resumed, i.e., the state of the `Suspendee` is changed from suspended into the state `Suspendee` was in before it was suspended.

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

Failures:

`badarg`
: If `Suspendee` isn't a process identifier.

`badarg`
: If the process calling `erlang:resume_process/1` had not previously increased the suspend count on the process identified by `Suspendee`.

`badarg`
: If the process identified by `Suspendee` is not alive.

`round(Number) -> int()`

Types:

Number = number()

Returns an integer by rounding `Number`.

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

Allowed in guard tests.

`self() -> pid()`

Returns the pid (process identifier) of the calling process.

```
> self().
<0.26.0>
```

Allowed in guard tests.

`erlang:send(Dest, Msg) -> Msg`

Types:

Dest = pid() | port() | RegName | {RegName, Node}

Msg = term()

RegName = atom()

Node = node()

Sends a message and returns `Msg`. This is the same as `Dest ! Msg`.

`Dest` may be a remote or local pid, a (local) port, a locally registered name, or a tuple `{RegName, Node}` for a registered name at another node.

`erlang:send(Dest, Msg, [Option]) -> Res`

Types:

Dest = pid() | port() | RegName | {RegName, Node}

RegName = atom()
Node = node()
Msg = term()
Option = nosuspend | noconnect
Res = ok | nosuspend | noconnect

Sends a message and returns `ok`, or does not send the message but returns something else (see below). Otherwise the same as *erlang:send/2*. See also *erlang:send_nosuspend/2,3*. for more detailed explanation and warnings.

The possible options are:

`nosuspend`

If the sender would have to be suspended to do the send, `nosuspend` is returned instead.

`noconnect`

If the destination node would have to be auto-connected before doing the send, `noconnect` is returned instead.

> Warning:
>
> As with `erlang:send_nosuspend/2,3`: Use with extreme care!

`erlang:send_after(Time, Dest, Msg) -> TimerRef`

Types:

Time = int()
0 <= Time <= 4294967295
Dest = pid() | RegName
LocalPid = pid() (of a process, alive or dead, on the local node)
Msg = term()
TimerRef = ref()

Starts a timer which will send the message `Msg` to `Dest` after `Time` milliseconds.

If `Dest` is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process.

If `Dest` is a pid, the timer will be automatically canceled if the process referred to by the pid is not alive, or when the process exits. This feature was introduced in erts version 5.4.11. Note that timers will not be automatically canceled when `Dest` is an atom.

See also *erlang:start_timer/3*, *erlang:cancel_timer/1*, and *erlang:read_timer/1*.

Failure: `badarg` if the arguments does not satisfy the requirements specified above.

`erlang:send_nosuspend(Dest, Msg) -> bool()`

Types:

Dest = pid() | port() | RegName | {RegName, Node}
RegName = atom()
Node = node()
Msg = term()

The same as *erlang:send(Dest, Msg, [nosuspend])*, but returns `true` if the message was sent and `false` if the message was not sent because the sender would have had to be suspended.

This function is intended for send operations towards an unreliable remote node without ever blocking the sending (Erlang) process. If the connection to the remote node (usually not a real Erlang node, but a node written in C or Java) is overloaded, this function *will not send the message* but return `false` instead.

The same happens, if `Dest` refers to a local port that is busy. For all other destinations (allowed for the ordinary send operator `'!'`) this function sends the message and returns `true`.

This function is only to be used in very rare circumstances where a process communicates with Erlang nodes that can disappear without any trace causing the TCP buffers and the drivers queue to be over-full before the node will actually be shut down (due to tick timeouts) by `net_kernel`. The normal reaction to take when this happens is some kind of premature shutdown of the other node.

Note that ignoring the return value from this function would result in *unreliable* message passing, which is contradictory to the Erlang programming model. The message is *not* sent if this function returns `false`.

Note also that in many systems, transient states of overloaded queues are normal. The fact that this function returns `false` does not in any way mean that the other node is guaranteed to be non-responsive, it could be a temporary overload. Also a return value of `true` does only mean that the message could be sent on the (TCP) channel without blocking, the message is not guaranteed to have arrived at the remote node. Also in the case of a disconnected non-responsive node, the return value is `true` (mimics the behaviour of the `!` operator). The expected behaviour as well as the actions to take when the function returns `false` are application and hardware specific.

> ### Warning:
>
> Use with extreme care!

`erlang:send_nosuspend(Dest, Msg, Options) -> bool()`

Types:

Dest = pid() | port() | RegName | {RegName, Node}

RegName = atom()

Node = node()

Msg = term()

Option = noconnect

The same as *erlang:send(Dest, Msg, [nosuspend | Options])*, but with boolean return value.

This function behaves like *erlang:send_nosuspend/2)*, but takes a third parameter, a list of options. The only currently implemented option is `noconnect`. The option `noconnect` makes the function return `false` if the remote node is not currently reachable by the local node. The normal behaviour is to try to connect to the node, which may stall the process for a shorter period. The use of the `noconnect` option makes it possible to be absolutely sure not to get even the slightest delay when sending to a remote process. This is especially useful when communicating with nodes who expect to always be the connecting part (i.e. nodes written in C or Java).

Whenever the function returns `false` (either when a suspend would occur or when `noconnect` was specified and the node was not already connected), the message is guaranteed *not* to have been sent.

> Warning:
>
> Use with extreme care!

`erlang:set_cookie(Node, Cookie) -> true`

Types:

Node = node()

Cookie = atom()

Sets the magic cookie of `Node` to the atom `Cookie`. If `Node` is the local node, the function also sets the cookie of all other unknown nodes to `Cookie` (see *Distributed Erlang* in the Erlang Reference Manual).

Failure: `function_clause` if the local node is not alive.

`setelement(Index, Tuple1, Value) -> Tuple2`

Types:

Index = 1..tuple_size(Tuple1)

Tuple1 = Tuple2 = tuple()

Value = term()

Returns a tuple which is a copy of the argument `Tuple1` with the element given by the integer argument `Index` (the first element is the element with index 1) replaced by the argument `Value`.

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

`size(Item) -> int()`

Types:

Item = tuple() | binary()

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

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

Allowed in guard tests.

`spawn(Fun) -> pid()`

Types:

Fun = fun()

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

`spawn(Node, Fun) -> pid()`

Types:

Node = node()

Fun = fun()

Returns the pid of a new process started by the application of `Fun` to the empty list `[]` on `Node`. If `Node` does not exist, a useless pid is returned. Otherwise works like *spawn/3*.

spawn(Module, Function, Args) -> pid()

Types:

Module = Function = atom()

Args = [term()]

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

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

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

spawn(Node, Module, Function, ArgumentList) -> pid()

Types:

Node = node()

Module = Function = atom()

Args = [term()]

Returns the pid of a new process started by the application of `Module:Function` to `Args` on `Node`. If `Node` does not exists, a useless pid is returned. Otherwise works like *spawn/3*.

spawn_link(Fun) -> pid()

Types:

Fun = fun()

Returns the pid of a new process started by the application of `Fun` to the empty list []. A link is created between the calling process and the new process, atomically. Otherwise works like *spawn/3*.

spawn_link(Node, Fun) -> pid()

Types:

Node = node()

Fun = fun()

Returns the pid of a new process started by the application of `Fun` to the empty list [] on `Node`. A link is created between the calling process and the new process, atomically. If `Node` does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason `noconnection` will be received). Otherwise works like *spawn/3*.

spawn_link(Module, Function, Args) -> pid()

Types:

Module = Function = atom()

Args = [term()]

Returns the pid of a new process started by the application of `Module:Function` to `Args`. A link is created between the calling process and the new process, atomically. Otherwise works like *spawn/3*.

`spawn_link(Node, Module, Function, Args) -> pid()`

Types:

Node = node()

Module = Function = atom()

Args = [term()]

Returns the pid of a new process started by the application of `Module:Function` to `Args` on `Node`. A link is created between the calling process and the new process, atomically. If `Node` does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason `noconnection` will be received). Otherwise works like *spawn/3*.

`spawn_monitor(Fun) -> {pid(),reference()}`

Types:

Fun = fun()

Returns the pid of a new process started by the application of `Fun` to the empty list [] and reference for a monitor created to the new process. Otherwise works like *spawn/3*.

`spawn_monitor(Module, Function, Args) -> {pid(),reference()}`

Types:

Module = Function = atom()

Args = [term()]

A new process is started by the application of `Module:Function` to `Args`, and the process is monitored at the same time. Returns the pid and a reference for the monitor. Otherwise works like *spawn/3*.

`spawn_opt(Fun, [Option]) -> pid() | {pid(),reference()}`

Types:

Fun = fun()

Option = link | monitor | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size} | {min_bin_vheap_size, VSize}

Level = low | normal | high

Number = int()

Size = int()

VSize = int()

Returns the pid of a new process started by the application of `Fun` to the empty list `[]`. Otherwise works like *spawn_opt/4*.

If the option `monitor` is given, the newly created process will be monitored and both the pid and reference for the monitor will be returned.

`spawn_opt(Node, Fun, [Option]) -> pid()`

Types:

Node = node()

Fun = fun()

Option = link | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size} | {min_bin_vheap_size, VSize}

Level = low | normal | high

Number = int()

Size = int()

VSize = int()

Returns the pid of a new process started by the application of `Fun` to the empty list `[]` on `Node`. If `Node` does not exist, a useless pid is returned. Otherwise works like *spawn_opt/4*.

`spawn_opt(Module, Function, Args, [Option]) -> pid() | {pid(),reference()}`

Types:

Module = Function = atom()

Args = [term()]

Option = link | monitor | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size} | {min_bin_vheap_size, VSize}

Level = low | normal | high

Number = int()

Size = int()

VSize = int()

Works exactly like *spawn/3*, except that an extra option list is given when creating the process.

If the option `monitor` is given, the newly created process will be monitored and both the pid and reference for the monitor will be returned.

`link`

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

`monitor`

Monitor the new process (just like *erlang:monitor/2* does).

`{priority, Level}`

Sets the priority of the new process. Equivalent to executing *process_flag(priority, Level)* in the start function of the new process, except that the priority will be set before the process is selected for execution for the first time. For more information on priorities see *process_flag(priority, Level)*.

`{fullsweep_after, Number}`

This option is only useful for performance tuning. In general, you should not use this option unless you know that there is problem with execution times and/or memory consumption, and you should measure to make sure that the option improved matters.

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

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

Here are a few cases when it could be useful to change `fullsweep_after`. Firstly, if binaries that are no longer used should be thrown away as soon as possible. (Set `Number` to zero.) Secondly, a process that mostly have short-lived data will be fullsweeped seldom or never, meaning that the old heap will contain mostly garbage.

To ensure a fullsweep once in a while, set `Number` to a suitable value such as 10 or 20. Thirdly, in embedded systems with limited amount of RAM and no virtual memory, one might want to preserve memory by setting `Number` to zero. (The value may be set globally, see *erlang:system_flag/2*.)

`{min_heap_size, Size}`

This option is only useful for performance tuning. In general, you should not use this option unless you know that there is problem with execution times and/or memory consumption, and you should measure to make sure that the option improved matters.

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

`{min_bin_vheap_size, VSize}`

This option is only useful for performance tuning. In general, you should not use this option unless you know that there is problem with execution times and/or memory consumption, and you should measure to make sure that the option improved matters.

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

`spawn_opt(Node, Module, Function, Args, [Option]) -> pid()`

Types:

Node = node()

Module = Function = atom()

Args = [term()]

Option = link | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size} | {min_bin_vheap_size, VSize}

Level = low | normal | high

Number = int()

Size = int()

VSize = int()

Returns the pid of a new process started by the application of `Module:Function` to `Args` on `Node`. If `Node` does not exist, a useless pid is returned. Otherwise works like *spawn_opt/4*.

`split_binary(Bin, Pos) -> {Bin1, Bin2}`

Types:

Bin = Bin1 = Bin2 = binary()

Pos = 0..byte_size(Bin)

Returns a tuple containing the binaries which are the result of splitting `Bin` into two parts at position `Pos`. This is not a destructive operation. After the operation, there will be three binaries altogether.

```
> B = list_to_binary("0123456789").
<<"0123456789">>
> byte_size(B).
10
```

```
> {B1, B2} = split_binary(B,3).
{<<"012">>,<<"3456789">>}
> byte_size(B1).
3
> byte_size(B2).
7
```

erlang:start_timer(Time, Dest, Msg) -> TimerRef

Types:

Time = int()
0 <= Time <= 4294967295
Dest = LocalPid | RegName
LocalPid = pid() (of a process, alive or dead, on the local node)
RegName = atom()
Msg = term()
TimerRef = ref()

Starts a timer which will send the message `{timeout, TimerRef, Msg}` to `Dest` after `Time` milliseconds.

If `Dest` is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process.

If `Dest` is a pid, the timer will be automatically canceled if the process referred to by the pid is not alive, or when the process exits. This feature was introduced in erts version 5.4.11. Note that timers will not be automatically canceled when `Dest` is an atom.

See also *erlang:send_after/3*, *erlang:cancel_timer/1*, and *erlang:read_timer/1*.

Failure: `badarg` if the arguments does not satisfy the requirements specified above.

statistics(Type) -> Res

Types:

Type, Res -- see below

Returns information about the system as specified by `Type`:

`context_switches`

Returns `{ContextSwitches, 0}`, where `ContextSwitches` is the total number of context switches since the system started.

`exact_reductions`

Returns `{Total_Exact_Reductions, Exact_Reductions_Since_Last_Call}`.

NOTE:`statistics(exact_reductions)` is a more expensive operation than *statistics(reductions)* especially on an Erlang machine with SMP support.

`garbage_collection`

Returns `{Number_of_GCs, Words_Reclaimed, 0}`. This information may not be valid for all implementations.

`io`

Returns `{{input, Input}, {output, Output}}`, where `Input` is the total number of bytes received through ports, and `Output` is the total number of bytes output to ports.

`reductions`

Returns `{Total_Reductions, Reductions_Since_Last_Call}`.

NOTE: From erts version 5.5 (OTP release R11B) this value does not include reductions performed in current time slices of currently scheduled processes. If an exact value is wanted, use *statistics(exact_reductions)*.

`run_queue`

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

`runtime`

Returns `{Total_Run_Time, Time_Since_Last_Call}`. Note that the run-time is the sum of the run-time for all threads in the Erlang run-time system and may therefore be greater than the wall-clock time.

`wall_clock`

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

All times are in milliseconds.

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

`erlang:suspend_process(Suspendee, OptList) -> true | false`

Types:

Suspendee = pid()

OptList = [Opt]

Opt = atom()

Increases the suspend count on the process identified by `Suspendee` and puts it in the suspended state if it isn't already in the suspended state. A suspended process will not be scheduled for execution until the process has been resumed.

A process can be suspended by multiple processes and can be suspended multiple times by a single process. A suspended process will not leave the suspended state until its suspend count reach zero. The suspend count of `Suspendee` is decreased when *erlang:resume_process(Suspendee)* is called by the same process that called `erlang:suspend_process(Suspendee)`. All increased suspend counts on other processes acquired by a process will automatically be decreased when the process terminates.

Currently the following options (`Opt`s) are available:

`asynchronous`

A suspend request is sent to the process identified by `Suspendee`. `Suspendee` will eventually suspend unless it is resumed before it was able to suspend. The caller of `erlang:suspend_process/2` will return immediately, regardless of whether the `Suspendee` has suspended yet or not. Note that the point in time when the `Suspendee` will actually suspend cannot be deduced from other events in the system. The only guarantee given is that the `Suspendee` will *eventually* suspend (unless it is resumed). If the `asynchronous` option has *not* been passed, the caller of `erlang:suspend_process/2` will be blocked until the `Suspendee` has actually suspended.

`unless_suspending`

The process identified by `Suspendee` will be suspended unless the calling process already is suspending the `Suspendee`. If `unless_suspending` is combined with the `asynchronous` option, a suspend request

will be sent unless the calling process already is suspending the `Suspendee` or if a suspend request already has been sent and is in transit. If the calling process already is suspending the `Suspendee`, or if combined with the `asynchronous` option and a send request already is in transit, `false` is returned and the suspend count on `Suspendee` will remain unchanged.

If the suspend count on the process identified by `Suspendee` was increased, `true` is returned; otherwise, `false` is returned.

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

Failures:

`badarg`
: If `Suspendee` isn't a process identifier.

`badarg`
: If the process identified by `Suspendee` is same the process as the process calling `erlang:suspend_process/2`.

`badarg`
: If the process identified by `Suspendee` is not alive.

`badarg`
: If the process identified by `Suspendee` resides on another node.

`badarg`
: If `OptList` isn't a proper list of valid `Opts`.

`system_limit`
: If the process identified by `Suspendee` has been suspended more times by the calling process than can be represented by the currently used internal data structures. The current system limit is larger than 2 000 000 000 suspends, and it will never be less than that.

`erlang:suspend_process(Suspendee) -> true`

Types:

Suspendee = pid()

Suspends the process identified by `Suspendee`. The same as calling *erlang:suspend_process(Suspendee, [])*. For more information see the documentation of *erlang:suspend_process/2*.

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

`erlang:system_flag(Flag, Value) -> OldValue`

Types:

Flag, Value, OldValue -- see below

Sets various system properties of the Erlang node. Returns the old value of the flag.

`erlang:system_flag(backtrace_depth, Depth)`

Sets the maximum depth of call stack back-traces in the exit reason element of `'EXIT'` tuples.

`erlang:system_flag(cpu_topology, CpuTopology)`

Sets the user defined `CpuTopology`. The user defined CPU topology will override any automatically detected CPU topology. By passing `undefined` as `CpuTopology` the system will revert back to the CPU topology automatically detected. The returned value equals the value returned from `erlang:system_info(cpu_topology)` before the change was made.

The CPU topology is used when binding schedulers to logical processors. If schedulers are already bound when the CPU topology is changed, the schedulers will be sent a request to rebind according to the new CPU topology.

The user defined CPU topology can also be set by passing the *+sct* command line argument to `erl`.

For information on the `CpuTopology` type and more, see the documentation of *erlang:system_info(cpu_topology)*, the `erl` *+sct* emulator flag, and *erlang:system_flag(scheduler_bind_type, How)*.

`erlang:system_flag(fullsweep_after, Number)`

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

In low-memory systems (especially without virtual memory), setting the value to 0 can help to conserve memory.

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

`erlang:system_flag(min_heap_size, MinHeapSize)`

Sets the default minimum heap size for processes. The size is given in words. The new `min_heap_size` only effects processes spawned after the change of `min_heap_size` has been made. The `min_heap_size` can be set for individual processes by use of *spawn_opt/N* or *process_flag/2*.

`erlang:system_flag(min_bin_vheap_size, MinBinVHeapSize)`

Sets the default minimum binary virtual heap size for processes. The size is given in words. The new `min_bin_vhheap_size` only effects processes spawned after the change of `min_bin_vhheap_size` has been made. The `min_bin_vheap_size` can be set for individual processes by use of *spawn_opt/N* or *process_flag/2*.

`erlang:system_flag(multi_scheduling, BlockState)`

`BlockState = block | unblock`

If multi-scheduling is enabled, more than one scheduler thread is used by the emulator. Multi-scheduling can be blocked. When multi-scheduling has been blocked, only one scheduler thread will schedule Erlang processes.

If `BlockState =:= block`, multi-scheduling will be blocked. If `BlockState =:= unblock` and no-one else is blocking multi-scheduling and this process has only blocked one time, multi-scheduling will be unblocked. One process can block multi-scheduling multiple times. If a process has blocked multiple times, it has to unblock exactly as many times as it has blocked before it has released its multi-scheduling block. If a process that has blocked multi-scheduling exits, it will release its blocking of multi-scheduling.

The return values are `disabled`, `blocked`, or `enabled`. The returned value describes the state just after the call to `erlang:system_flag(multi_scheduling, BlockState)` has been made. The return values are described in the documentation of *erlang:system_info(multi_scheduling)*.

NOTE: Blocking of multi-scheduling should normally not be needed. If you feel that you need to block multi-scheduling, think through the problem at least a couple of times again. Blocking multi-scheduling should only be used as a last resort since it will most likely be a *very inefficient* way to solve the problem.

See also *erlang:system_info(multi_scheduling)*, *erlang:system_info(multi_scheduling_blockers)*, and *erlang:system_info(schedulers)*.

`erlang:system_flag(scheduler_bind_type, How)`

Controls if and how schedulers are bound to logical processors.

When `erlang:system_flag(scheduler_bind_type, How)` is called, an asynchronous signal is sent to all schedulers online which causes them to try to bind or unbind as requested. *NOTE:* If a scheduler fails to bind, this will often be silently ignored. This since it isn't always possible to verify valid logical processor identifiers. If an error is reported, it will be reported to the `error_logger`. If you want to verify that the schedulers actually have bound as requested, call *erlang:system_info(scheduler_bindings)*.

Schedulers can currently only be bound on newer Linux and Solaris systems, but more systems will be supported in the future.

In order for the runtime system to be able to bind schedulers, the CPU topology needs to be known. If the runtime system fails to automatically detect the CPU topology, it can be defined. For more information on how to define the CPU topology, see *erlang:system_flag(cpu_topology, CpuTopology)*.

NOTE: If other programs on the system have bound to processors, e.g. another Erlang runtime system, you may loose performance when binding schedulers. Therefore, schedulers are by default not bound.

Schedulers can be bound in different ways. The `How` argument determines how schedulers are bound. `How` can currently be one of:

`unbound`

Schedulers will not be bound to logical processors, i.e., the operating system decides where the scheduler threads execute, and when to migrate them. This is the default.

`no_spread`

Schedulers with close scheduler identifiers will be bound as close as possible in hardware.

`thread_spread`

Thread refers to hardware threads (e.g. Intels hyper-threads). Schedulers with low scheduler identifiers, will be bound to the first hardware thread of each core, then schedulers with higher scheduler identifiers will be bound to the second hardware thread of each core, etc.

`processor_spread`

Schedulers will be spread like `thread_spread`, but also over physical processor chips.

`spread`

Schedulers will be spread as much as possible.

`no_node_thread_spread`

Like `thread_spread`, but if multiple NUMA (Non-Uniform Memory Access) nodes exists, schedulers will be spread over one NUMA node at a time, i.e., all logical processors of one NUMA node will be bound to schedulers in sequence.

`no_node_processor_spread`

Like `processor_spread`, but if multiple NUMA nodes exists, schedulers will be spread over one NUMA node at a time, i.e., all logical processors of one NUMA node will be bound to schedulers in sequence.

`thread_no_node_processor_spread`

A combination of `thread_spread`, and `no_node_processor_spread`. Schedulers will be spread over hardware threads across NUMA nodes, but schedulers will only be spread over processors internally in one NUMA node at a time.

`default_bind`

Binds schedulers the default way. Currently the default is `thread_no_node_processor_spread` (which might change in the future).

How schedulers are bound matters. For example, in situations when there are fewer running processes than schedulers online, the runtime system tries to migrate processes to schedulers with low scheduler identifiers. The more the schedulers are spread over the hardware, the more resources will be available to the runtime system in such situations.

The value returned equals `How` before the `scheduler_bind_type` flag was changed.

Failure:

`notsup`

If binding of schedulers is not supported.

`badarg`

If `How` isn't one of the documented alternatives.

`badarg`

If no CPU topology information is available.

The scheduler bind type can also be set by passing the *+sbt* command line argument to `erl`.

For more information, see *erlang:system_info(scheduler_bind_type)*, *erlang:system_info(scheduler_bindings)*, the `erl` *+sbt* emulator flag, and *erlang:system_flag(cpu_topology, CpuTopology)*.

`erlang:system_flag(schedulers_online, SchedulersOnline)`

Sets the amount of schedulers online. Valid range is 1 <= SchedulerId <= erlang:system_info(schedulers).

For more information see, *erlang:system_info(schedulers)*, and *erlang:system_info(schedulers_online)*.

`erlang:system_flag(trace_control_word, TCW)`

Sets the value of the node's trace control word to `TCW`. `TCW` should be an unsigned integer. For more information see documentation of the *set_tcw* function in the match specification documentation in the ERTS User's Guide.

> ### Note:
>
> The `schedulers` option has been removed as of erts version 5.5.3. The number of scheduler threads is determined at emulator boot time, and cannot be changed after that.

`erlang:system_info(Type) -> Res`

Types:

Type, Res -- see below

Returns various information about the current system (emulator) as specified by `Type`:

`allocated_areas`

Returns a list of tuples with information about miscellaneous allocated memory areas.

Each tuple contains an atom describing type of memory as first element and amount of allocated memory in bytes as second element. In those cases when there is information present about allocated and used memory, a third element is present. This third element contains the amount of used memory in bytes.

`erlang:system_info(allocated_areas)` is intended for debugging, and the content is highly implementation dependent. The content of the results will therefore change when needed without prior notice.

Note: The sum of these values is *not* the total amount of memory allocated by the emulator. Some values are part of other values, and some memory areas are not part of the result. If you are interested in the total amount of memory allocated by the emulator see *erlang:memory/0,1*.

`allocator`

Returns `{Allocator, Version, Features, Settings}`.

Types:

- `Allocator = undefined | elib_malloc | glibc`
- `Version = [int()]`
- `Features = [atom()]`
- `Settings = [{Subsystem, [{Parameter, Value}]}]`
- `Subsystem = atom()`
- `Parameter = atom()`
- `Value = term()`

Explanation:

- `Allocator` corresponds to the `malloc()` implementation used. If `Allocator` equals `undefined`, the `malloc()` implementation used could not be identified. Currently `elib_malloc` and `glibc` can be identified.
- `Version` is a list of integers (but not a string) representing the version of the `malloc()` implementation used.
- `Features` is a list of atoms representing allocation features used.
- `Settings` is a list of subsystems, their configurable parameters, and used values. Settings may differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes.

See also "System Flags Effecting erts_alloc" in *erts_alloc(3)*.

`alloc_util_allocators`

Returns a list of the names of all allocators using the ERTS internal `alloc_util` framework as atoms. For more information see the *"the alloc_util framework" section in the erts_alloc(3)* documentation.

`{allocator, Alloc}`

Returns information about the specified allocator. As of erts version 5.6.1 the return value is a list of `{instance, InstanceNo, InstanceInfo}` tuples where `InstanceInfo` contains information about a specific instance of the allocator. If `Alloc` is not a recognized allocator, `undefined` is returned. If `Alloc` is disabled, `false` is returned.

Note: The information returned is highly implementation dependent and may be changed, or removed at any time without prior notice. It was initially intended as a tool when developing new allocators, but since it might be of interest for others it has been briefly documented.

The recognized allocators are listed in *erts_alloc(3)*. After reading the `erts_alloc(3)` documentation, the returned information should more or less speak for itself. But it can be worth explaining some things. Call counts are presented by two values. The first value is giga calls, and the second value is calls. `mbcs`, and `sbcs` are abbreviations for, respectively, multi-block carriers, and single-block carriers. Sizes are presented in bytes. When it is not a size that is presented, it is the amount of something. Sizes and amounts are often presented by three values, the first is current value, the second is maximum value since the last call to `erlang:system_info({allocator, Alloc})`, and the third is maximum value since the emulator was

started. If only one value is present, it is the current value. `fix_alloc` memory block types are presented by two values. The first value is memory pool size and the second value used memory size.

`{allocator_sizes, Alloc}`

Returns various size information for the specified allocator. The information returned is a subset of the information returned by *erlang:system_info({allocator, Alloc})*.

`c_compiler_used`

Returns a two-tuple describing the C compiler used when compiling the runtime system. The first element is an atom describing the name of the compiler, or `undefined` if unknown. The second element is a term describing the version of the compiler, or `undefined` if unknown.

`check_io`

Returns a list containing miscellaneous information regarding the emulators internal I/O checking. Note, the content of the returned list may vary between platforms and over time. The only thing guaranteed is that a list is returned.

`compat_rel`

Returns the compatibility mode of the local node as an integer. The integer returned represents the Erlang/OTP release which the current emulator has been set to be backward compatible with. The compatibility mode can be configured at startup by using the command line flag `+R`, see *erl(1)*.

`cpu_topology`

Returns the `CpuTopology` which currently is used by the emulator. The CPU topology is used when binding schedulers to logical processors. The CPU topology used is the user defined CPU topology if such exist; otherwise, the automatically detected CPU topology if such exist. If no CPU topology exist `undefined` is returned.

Types:

- `CpuTopology = LevelEntryList | undefined`
- `LevelEntryList = [LevelEntry]` (all `LevelEntry`s of a `LevelEntryList` must contain the same `LevelTag`, except on the top level where both `node` and `processor` `LevelTag`s may co-exist)
- `LevelEntry = {LevelTag, SubLevel} | {LevelTag, InfoList, SubLevel}` (`{LevelTag, SubLevel} == {LevelTag, [], SubLevel}`)
- `LevelTag = node|processor|core|thread` (more `LevelTag`s may be introduced in the future)
- `SubLevel = [LevelEntry] | LogicalCpuId`
- `LogicalCpuId = {logical, integer()}`
- `InfoList = []` (the `InfoList` may be extended in the future)

`node` refers to NUMA (non-uniform memory access) nodes, and `thread` refers to hardware threads (e.g. Intels hyper-threads).

A level in the `CpuTopology` term can be omitted if only one entry exists and the `InfoList` is empty.

`thread` can only be a sub level to `core`. `core` can be a sub level to either `processor` or `node`. `processor` can either be on the top level or a sub level to `node`. `node` can either be on the top level or a sub level to `processor`. That is, NUMA nodes can be processor internal or processor external. A CPU topology can consist of a mix of processor internal and external NUMA nodes, as long as each logical CPU belongs to one and only one NUMA node. Cache hierarchy is not part of the `CpuTopology` type yet, but will be in the future. Other things may also make it into the CPU topology in the future. In other words, expect the `CpuTopology` type to change.

`{cpu_topology, defined}`

Returns the user defined `CpuTopology`. For more information see the documentation of *erlang:system_flag(cpu_topology, CpuTopology)* and the documentation of the *cpu_topology* argument.

`{cpu_topology, detected}`

Returns the automatically detected `CpuTopology`. The emulator currently only detects the CPU topology on some newer linux and solaris systems. For more information see the documentation of the *cpu_topology* argument.

`{cpu_topology, used}`

Returns the `CpuTopology` which is used by the emulator. For more information see the documentation of the *cpu_topology* argument.

`creation`

Returns the creation of the local node as an integer. The creation is changed when a node is restarted. The creation of a node is stored in process identifiers, port identifiers, and references. This makes it (to some extent) possible to distinguish between identifiers from different incarnations of a node. Currently valid creations are integers in the range 1..3, but this may (probably will) change in the future. If the node is not alive, 0 is returned.

`debug_compiled`

Returns `true` if the emulator has been debug compiled; otherwise, `false`.

`dist`

Returns a binary containing a string of distribution information formatted as in Erlang crash dumps. For more information see the *"How to interpret the Erlang crash dumps"* chapter in the ERTS User's Guide.

`dist_ctrl`

Returns a list of tuples `{Node, ControllingEntity}`, one entry for each connected remote node. The `Node` is the name of the node and the `ControllingEntity` is the port or pid responsible for the communication to that node. More specifically, the `ControllingEntity` for nodes connected via TCP/IP (the normal case) is the socket actually used in communication with the specific node.

`driver_version`

Returns a string containing the erlang driver version used by the runtime system. It will be on the form *"<major ver>.<minor ver>"*.

`elib_malloc`

If the emulator uses the `elib_malloc` memory allocator, a list of two-element tuples containing status information is returned; otherwise, `false` is returned. The list currently contains the following two-element tuples (all sizes are presented in bytes):

`{heap_size, Size}`

Where `Size` is the current heap size.

`{max_alloced_size, Size}`

Where `Size` is the maximum amount of memory allocated on the heap since the emulator started.

`{alloced_size, Size}`

Where `Size` is the current amount of memory allocated on the heap.

`{free_size, Size}`

Where `Size` is the current amount of free memory on the heap.

`{no_alloced_blocks, No}`

Where `No` is the current number of allocated blocks on the heap.

`{no_free_blocks, No}`

Where `No` is the current number of free blocks on the heap.

`{smallest_alloced_block, Size}`

Where `Size` is the size of the smallest allocated block on the heap.

`{largest_free_block, Size}`

Where `Size` is the size of the largest free block on the heap.

`fullsweep_after`

Returns `{fullsweep_after, int()}` which is the `fullsweep_after` garbage collection setting used by default. For more information see `garbage_collection` described below.

`garbage_collection`

Returns a list describing the default garbage collection settings. A process spawned on the local node by a `spawn` or `spawn_link` will use these garbage collection settings. The default settings can be changed by use of *system_flag/2*. *spawn_opt/4* can spawn a process that does not use the default settings.

`global_heaps_size`

Returns the current size of the shared (global) heap.

`heap_sizes`

Returns a list of integers representing valid heap sizes in words. All Erlang heaps are sized from sizes in this list.

`heap_type`

Returns the heap type used by the current emulator. Currently the following heap types exist:

`private`

Each process has a heap reserved for its use and no references between heaps of different processes are allowed. Messages passed between processes are copied between heaps.

`shared`

One heap for use by all processes. Messages passed between processes are passed by reference.

`hybrid`

A hybrid of the `private` and `shared` heap types. A shared heap as well as private heaps are used.

`info`

Returns a binary containing a string of miscellaneous system information formatted as in Erlang crash dumps. For more information see the *"How to interpret the Erlang crash dumps"* chapter in the ERTS User's Guide.

`kernel_poll`

Returns `true` if the emulator uses some kind of kernel-poll implementation; otherwise, `false`.

`loaded`

Returns a binary containing a string of loaded module information formatted as in Erlang crash dumps. For more information see the *"How to interpret the Erlang crash dumps"* chapter in the ERTS User's Guide.

`logical_processors`

Returns the number of logical processors detected on the system as an integer or the atom `unknown` if the emulator wasn't able to detect any.

`machine`

Returns a string containing the Erlang machine name.

`min_heap_size`

Returns `{min_heap_size, MinHeapSize}` where `MinHeapSize` is the current system wide minimum heap size for spawned processes.

`min_bin_vheap_size`

Returns `{min_bin_vheap_size, MinBinVHeapSize}` where `MinBinVHeapSize` is the current system wide minimum binary virtual heap size for spawned processes.

`modified_timing_level`

Returns the modified timing level (an integer) if modified timing has been enabled; otherwise, `undefined`. See the `+T` command line flag in the documentation of the *erl(1)* command for more information on modified timing.

`multi_scheduling`

Returns `disabled`, `blocked`, or `enabled`. A description of the return values:

`disabled`

The emulator has only one scheduler thread. The emulator does not have SMP support, or have been started with only one scheduler thread.

`blocked`

The emulator has more than one scheduler thread, but all scheduler threads but one have been blocked, i.e., only one scheduler thread will schedule Erlang processes and execute Erlang code.

`enabled`

The emulator has more than one scheduler thread, and no scheduler threads have been blocked, i.e., all available scheduler threads will schedule Erlang processes and execute Erlang code.

See also *erlang:system_flag(multi_scheduling, BlockState)*, *erlang:system_info(multi_scheduling_blockers)*, and *erlang:system_info(schedulers)*.

`multi_scheduling_blockers`

Returns a list of `PID`s when multi-scheduling is blocked; otherwise, the empty list. The `PID`s in the list is `PID`s of the processes currently blocking multi-scheduling. A `PID` will only be present once in the list, even if the corresponding process has blocked multiple times.

See also *erlang:system_flag(multi_scheduling, BlockState)*, *erlang:system_info(multi_scheduling)*, and *erlang:system_info(schedulers)*.

`otp_release`

Returns a string containing the OTP release number.

`process_count`

Returns the number of processes currently existing at the local node as an integer. The same value as `length(processes())` returns.

`process_limit`

Returns the maximum number of concurrently existing processes at the local node as an integer. This limit can be configured at startup by using the command line flag `+P`, see *erl(1)*.

`procs`

Returns a binary containing a string of process and port information formatted as in Erlang crash dumps. For more information see the *"How to interpret the Erlang crash dumps"* chapter in the ERTS User's Guide.

`scheduler_bind_type`

Returns information on how user has requested schedulers to be bound or not bound.

NOTE: Even though user has requested schedulers to be bound via *erlang:system_flag(scheduler_bind_type, How)*, they might have silently failed to bind. In order to inspect actual scheduler bindings call *erlang:system_info(scheduler_bindings)*.

For more information, see *erlang:system_flag(scheduler_bind_type, How)*, and *erlang:system_info(scheduler_bindings)*.

`scheduler_bindings`

Returns information on currently used scheduler bindings.

A tuple of a size equal to *erlang:system_info(schedulers)* is returned. The elements of the tuple are integers or the atom `unbound`. Logical processor identifiers are represented as integers. The `N`th element of the tuple equals the current binding for the scheduler with the scheduler identifier equal to `N`. E.g., if the schedulers have been bound, `element(erlang:system_info(scheduler_id), erlang:system_info(scheduler_bindings))` will return the identifier of the logical processor that the calling process is executing on.

Note that only schedulers online can be bound to logical processors.

For more information, see *erlang:system_flag(scheduler_bind_type, How)*, *erlang:system_info(schedulers_online)*.

`scheduler_id`

Returns the scheduler id (`SchedulerId`) of the scheduler thread that the calling process is executing on. `SchedulerId` is a positive integer; where `1 <= SchedulerId <= erlang:system_info(schedulers)`. See also *erlang:system_info(schedulers)*.

`schedulers`

Returns the number of scheduler threads used by the emulator. Scheduler threads online schedules Erlang processes and Erlang ports, and execute Erlang code and Erlang linked in driver code.

The number of scheduler threads is determined at emulator boot time and cannot be changed after that. The amount of schedulers online can however be changed at any time.

See also *erlang:system_flag(schedulers_online, SchedulersOnline)*, *erlang:system_info(schedulers_online)*, *erlang:system_info(scheduler_id)*, *erlang:system_flag(multi_scheduling, BlockState)*, *erlang:system_info(multi_scheduling)*, and and *erlang:system_info(multi_scheduling_blockers)*.

`schedulers_online`

Returns the amount of schedulers online. The scheduler identifiers of schedulers online satisfy the following relationship: `1 <= SchedulerId <= erlang:system_info(schedulers_online)`.

For more information, see *erlang:system_info(schedulers)*, and *erlang:system_flag(schedulers_online, SchedulersOnline)*.

`smp_support`

Returns `true` if the emulator has been compiled with smp support; otherwise, `false`.

`system_version`

Returns a string containing version number and some important properties such as the number of schedulers.

`system_architecture`

Returns a string containing the processor and OS architecture the emulator is built for.

`threads`

Returns `true` if the emulator has been compiled with thread support; otherwise, `false` is returned.

`thread_pool_size`

Returns the number of async threads in the async thread pool used for asynchronous driver calls (*driver_async()*) as an integer.

`trace_control_word`

Returns the value of the node's trace control word. For more information see documentation of the function `get_tcw` in "Match Specifications in Erlang", *ERTS User's Guide*.

`version`

Returns a string containing the version number of the emulator.

`wordsize`

Returns the word size in bytes as an integer, i.e. on a 32-bit architecture 4 is returned, and on a 64-bit architecture 8 is returned.

> Note:
>
> The `scheduler` argument has changed name to `scheduler_id`. This in order to avoid mixup with the `schedulers` argument. The `scheduler` argument was introduced in ERTS version 5.5 and renamed in ERTS version 5.5.1.

`erlang:system_monitor() -> MonSettings`

Types:

MonSettings -> {MonitorPid, Options} | undefined
MonitorPid = pid()
Options = [Option]
Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
Time = Size = int()

Returns the current system monitoring settings set by *erlang:system_monitor/2* as `{MonitorPid, Options}`, or `undefined` if there are no settings. The order of the options may be different from the one that was set.

`erlang:system_monitor(undefined | {MonitorPid, Options}) -> MonSettings`

Types:

MonitorPid, Options, MonSettings -- see below

When called with the argument `undefined`, all system performance monitoring settings are cleared.

Calling the function with `{MonitorPid, Options}` as argument, is the same as calling *erlang:system_monitor(MonitorPid, Options)*.

Returns the previous system monitor settings just like *erlang:system_monitor/0*.

`erlang:system_monitor(MonitorPid, [Option]) -> MonSettings`

Types:

MonitorPid = pid()
Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
Time = Size = int()
MonSettings = {OldMonitorPid, [Option]}

OldMonitorPid = pid()

Sets system performance monitoring options. `MonitorPid` is a local pid that will receive system monitor messages, and the second argument is a list of monitoring options:

`{long_gc, Time}`

If a garbage collection in the system takes at least `Time` wallclock milliseconds, a message `{monitor, GcPid, long_gc, Info}` is sent to `MonitorPid`. `GcPid` is the pid that was garbage collected and `Info` is a list of two-element tuples describing the result of the garbage collection. One of the tuples is `{timeout, GcTime}` where `GcTime` is the actual time for the garbage collection in milliseconds. The other tuples are tagged with `heap_size`, `heap_block_size`, `stack_size`, `mbuf_size`, `old_heap_size`, and `old_heap_block_size`. These tuples are explained in the documentation of the *gc_start* trace message (see *erlang:trace/3*). New tuples may be added, and the order of the tuples in the `Info` list may be changed at any time without prior notice.

`{large_heap, Size}`

If a garbage collection in the system results in the allocated size of a heap being at least `Size` words, a message `{monitor, GcPid, large_heap, Info}` is sent to `MonitorPid`. `GcPid` and `Info` are the same as for `long_gc` above, except that the tuple tagged with `timeout` is not present. *Note*: As of erts version 5.6 the monitor message is sent if the sum of the sizes of all memory blocks allocated for all heap generations is equal to or larger than `Size`. Previously the monitor message was sent if the memory block allocated for the youngest generation was equal to or larger than `Size`.

`busy_port`

If a process in the system gets suspended because it sends to a busy port, a message `{monitor, SusPid, busy_port, Port}` is sent to `MonitorPid`. `SusPid` is the pid that got suspended when sending to `Port`.

`busy_dist_port`

If a process in the system gets suspended because it sends to a process on a remote node whose inter-node communication was handled by a busy port, a message `{monitor, SusPid, busy_dist_port, Port}` is sent to `MonitorPid`. `SusPid` is the pid that got suspended when sending through the inter-node communication port `Port`.

Returns the previous system monitor settings just like *erlang:system_monitor/0*.

> ### Note:
>
> If a monitoring process gets so large that it itself starts to cause system monitor messages when garbage collecting, the messages will enlarge the process's message queue and probably make the problem worse.
>
> Keep the monitoring process neat and do not set the system monitor limits too tight.

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

`erlang:system_profile() -> ProfilerSettings`

Types:

ProfilerSettings -> {ProfilerPid, Options} | undefined

ProfilerPid = pid() | port()

Options = [Option]

Option = runnable_procs | runnable_ports | scheduler | exclusive

Returns the current system profiling settings set by *erlang:system_profile/2* as `{ProfilerPid, Options}`, or `undefined` if there are no settings. The order of the options may be different from the one that was set.

`erlang:system_profile(ProfilerPid, Options) -> ProfilerSettings`

Types:

ProfilerSettings -> {ProfilerPid, Options} | undefined

ProfilerPid = pid() | port()

Options = [Option]

Option = runnable_procs | runnable_ports | scheduler | exclusive

Sets system profiler options. `ProfilerPid` is a local pid or port that will receive profiling messages. The receiver is excluded from all profiling. The second argument is a list of profiling options:

`runnable_procs`

If a process is put into or removed from the run queue a message, `{profile, Pid, State, Mfa, Ts}`, is sent to `ProfilerPid`. Running processes that is reinserted into the run queue after having been preemptively scheduled out will not trigger this message.

`runnable_ports`

If a port is put into or removed from the run queue a message, `{profile, Port, State, 0, Ts}`, is sent to `ProfilerPid`.

`scheduler`

If a scheduler is put to sleep or awoken a message, `{profile, scheduler, Id, State, NoScheds, Ts}`, is sent to `ProfilerPid`.

`exclusive`

If a synchronous call to a port from a process is done, the calling process is considered not runnable during the call runtime to the port. The calling process is notified as `inactive` and subsequently `active` when the port callback returns.

> Note:
>
> `erlang:system_profile` is considered experimental and its behaviour may change in the future.

`term_to_binary(Term) -> ext_binary()`

Types:

Term = term()

Returns a binary data object which is the result of encoding `Term` according to the Erlang external term format.

This can be used for a variety of purposes, for example writing a term to a file in an efficient way, or sending an Erlang term to some type of communications channel not supported by distributed Erlang.

See also *binary_to_term/1*.

`term_to_binary(Term, [Option]) -> ext_binary()`

Types:

Term = term()

Option = compressed | {compressed,Level} | {minor_version,Version}

Returns a binary data object which is the result of encoding `Term` according to the Erlang external term format.

If the option `compressed` is provided, the external term format will be compressed. The compressed format is automatically recognized by `binary_to_term/1` in R7B and later.

It is also possible to specify a compression level by giving the option `{compressed,Level}`, where `Level` is an integer from 0 through 9. `0` means that no compression will be done (it is the same as not giving any `compressed` option); `1` will take the least time but may not compress as well as the higher levels; `9` will take the most time and may produce a smaller result. Note the "mays" in the preceding sentence; depending on the input term, level 9 compression may or may not produce a smaller result than level 1 compression.

Currently, `compressed` gives the same result as `{compressed,6}`.

The option `{minor_version,Version}` can be use to control some details of the encoding. This option was introduced in R11B-4. Currently, the allowed values for `Version` are `0` and `1`.

`{minor_version,1}` forces any floats in the term to be encoded in a more space-efficient and exact way (namely in the 64-bit IEEE format, rather than converted to a textual representation). `binary_to_term/1` in R11B-4 and later is able decode the new representation.

`{minor_version,0}` is currently the default, meaning that floats will be encoded using a textual representation; this option is useful if you want to ensure that releases prior to R11B-4 can decode resulting binary.

See also *binary_to_term/1*.

`throw(Any)`

Types:

Any = term()

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

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

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

`time() -> {Hour, Minute, Second}`

Types:

Hour = Minute = Second = int()

Returns the current time as `{Hour, Minute, Second}`.

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

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

`tl(List1) -> List2`

Types:

List1 = List2 = [term()]

Returns the tail of `List1`, that is, the list minus the first element.

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

Allowed in guard tests.

Failure: `badarg` if `List` is the empty list [].

`erlang:trace(PidSpec, How, FlagList) -> int()`

Types:

PidSpec = pid() | existing | new | all

How = bool()

FlagList = [Flag]

Flag -- see below

Turns on (if `How == true`) or off (if `How == false`) the trace flags in `FlagList` for the process or processes represented by `PidSpec`.

`PidSpec` is either a pid for a local process, or one of the following atoms:

`existing`

All processes currently existing.

`new`

All processes that will be created in the future.

`all`

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

`FlagList` can contain any number of the following flags (the "message tags" refers to the list of messages following below):

`all`

Set all trace flags except `{tracer, Tracer}` and `cpu_timestamp` that are in their nature different than the others.

`send`

Trace sending of messages.

Message tags: `send`, `send_to_non_existing_process`.

`'receive'`

Trace receiving of messages.

Message tags: `'receive'`.

`procs`

Trace process related events.

Message tags: `spawn`, `exit`, `register`, `unregister`, `link`, `unlink`, `getting_linked`, `getting_unlinked`.

`call`

Trace certain function calls. Specify which function calls to trace by calling *erlang:trace_pattern/3*.

Message tags: `call`, `return_from`.

`silent`

Used in conjunction with the `call` trace flag. The `call`, `return_from` and `return_to` trace messages are inhibited if this flag is set, but if there are match specs they are executed as normal.

Silent mode is inhibited by executing `erlang:trace(_, false, [silent|_])`, or by a match spec executing the `{silent, false}` function.

The `silent` trace flag facilitates setting up a trace on many or even all processes in the system. Then the interesting trace can be activated and deactivated using the `{silent,Bool}` match spec function, giving a high degree of control of which functions with which arguments that triggers the trace.

Message tags: `call`, `return_from`, `return_to`. Or rather, the absence of.

`return_to`

Used in conjunction with the `call` trace flag. Trace the actual return from a traced function back to its caller. Only works for functions traced with the `local` option to *erlang:trace_pattern/3*.

The semantics is that a trace message is sent when a call traced function actually returns, that is, when a chain of tail recursive calls is ended. There will be only one trace message sent per chain of tail recursive calls, why the properties of tail recursiveness for function calls are kept while tracing with this flag. Using `call` and `return_to` trace together makes it possible to know exactly in which function a process executes at any time.

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

Message tags: `return_to`.

`running`

Trace scheduling of processes.

Message tags: `in`, and `out`.

`exiting`

Trace scheduling of an exiting processes.

Message tags: `in_exiting`, `out_exiting`, and `out_exited`.

`garbage_collection`

Trace garbage collections of processes.

Message tags: `gc_start`, `gc_end`.

`timestamp`

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

`cpu_timestamp`

A global trace flag for the Erlang node that makes all trace timestamps be in CPU time, not wallclock. It is only allowed with `PidSpec==all`. If the host machine operating system does not support high resolution CPU time measurements, `trace/3` exits with `badarg`.

`arity`

Used in conjunction with the `call` trace flag. `{M, F, Arity}` will be specified instead of `{M, F, Args}` in call trace messages.

`set_on_spawn`

Makes any process created by a traced process inherit its trace flags, including the `set_on_spawn` flag.

`set_on_first_spawn`

Makes the first process created by a traced process inherit its trace flags, excluding the `set_on_first_spawn` flag.

`set_on_link`

Makes any process linked by a traced process inherit its trace flags, including the `set_on_link` flag.

`set_on_first_link`

Makes the first process linked to by a traced process inherit its trace flags, excluding the `set_on_first_link` flag.

`{tracer, Tracer}`

Specify where to send the trace messages. `Tracer` must be the pid of a local process or the port identifier of a local port. If this flag is not given, trace messages will be sent to the process that called `erlang:trace/3`.

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

If the `timestamp` flag is not given, the tracing process will receive the trace messages described below. `Pid` is the pid of the traced process in which the traced event has occurred. The third element of the tuple is the message tag.

If the `timestamp` flag is given, the first element of the tuple will be `trace_ts` instead and the timestamp is added last in the tuple.

`{trace, Pid, 'receive', Msg}`

When `Pid` receives the message `Msg`.

`{trace, Pid, send, Msg, To}`

When `Pid` sends the message `Msg` to the process `To`.

`{trace, Pid, send_to_non_existing_process, Msg, To}`

When `Pid` sends the message `Msg` to the non-existing process `To`.

`{trace, Pid, call, {M, F, Args}}`

When `Pid` calls a traced function. The return values of calls are never supplied, only the call and its arguments.

Note that the trace flag `arity` can be used to change the contents of this message, so that `Arity` is specified instead of `Args`.

`{trace, Pid, return_to, {M, F, Arity}}`

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

`{trace, Pid, return_from, {M, F, Arity}, ReturnValue}`

When `Pid` returns *from* the specified function. This trace message is sent if the `call` flag is set, and the function has a match specification with a `return_trace` or `exception_trace` action.

`{trace, Pid, exception_from, {M, F, Arity}, {Class, Value}}`

When `Pid` exits *from* the specified function due to an exception. This trace message is sent if the `call` flag is set, and the function has a match specification with an `exception_trace` action.

`{trace, Pid, spawn, Pid2, {M, F, Args}}`

When `Pid` spawns a new process `Pid2` with the specified function call as entry point.

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

`{trace, Pid, exit, Reason}`

When `Pid` exits with reason `Reason`.

`{trace, Pid, link, Pid2}`

When `Pid` links to a process `Pid2`.

`{trace, Pid, unlink, Pid2}`

When `Pid` removes the link from a process `Pid2`.

`{trace, Pid, getting_linked, Pid2}`

When `Pid` gets linked to a process `Pid2`.

`{trace, Pid, getting_unlinked, Pid2}`

When `Pid` gets unlinked from a process `Pid2`.

`{trace, Pid, register, RegName}`

When `Pid` gets the name `RegName` registered.

`{trace, Pid, unregister, RegName}`

When `Pid` gets the name `RegName` unregistered. Note that this is done automatically when a registered process exits.

`{trace, Pid, in, {M, F, Arity} | 0}`

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

`{trace, Pid, out, {M, F, Arity} | 0}`

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

`{trace, Pid, gc_start, Info}`

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

`heap_size`
The size of the used part of the heap.
`heap_block_size`
The size of the memory block used for storing the heap and the stack.
`old_heap_size`
The size of the used part of the old heap.
`old_heap_block_size`
The size of the memory block used for storing the old heap.
`stack_size`
The actual size of the stack.
`recent_size`
The size of the data that survived the previous garbage collection.
`mbuf_size`
The combined size of message buffers associated with the process.
`bin_vheap_size`
The total size of unique off-heap binaries referenced from the process heap.

`bin_vheap_block_size`
The total size of binaries, in words, allowed in the virtual heap in the process before doing a garbage collection.
`bin_old_vheap_size`
The total size of unique off-heap binaries referenced from the process old heap.
`bin_vheap_block_size`
The total size of binaries, in words, allowed in the virtual old heap in the process before doing a garbage collection.

All sizes are in words.

`{trace, Pid, gc_end, Info}`

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

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

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

Returns: A number indicating the number of processes that matched `PidSpec`. If `PidSpec` is a pid, the return value will be `1`. If `PidSpec` is `all` or `existing` the return value will be the number of processes running, excluding tracer processes. If `PidSpec` is `new`, the return value will be `0`.

Failure: If specified arguments are not supported. For example `cpu_timestamp` is not supported on all platforms.

`erlang:trace_delivered(Tracee) -> Ref`

Types:

Tracee = pid() | all

Ref = reference()

The delivery of trace messages is dislocated on the time-line compared to other events in the system. If you know that the `Tracee` has passed some specific point in its execution, and you want to know when at least all trace messages corresponding to events up to this point have reached the tracer you can use `erlang:trace_delivered(Tracee)`. A `{trace_delivered, Tracee, Ref}` message is sent to the caller of `erlang:trace_delivered(Tracee)` when it is guaranteed that all trace messages have been delivered to the tracer up to the point that the `Tracee` had reached at the time of the call to `erlang:trace_delivered(Tracee)`.

Note that the `trace_delivered` message does *not* imply that trace messages have been delivered; instead, it implies that all trace messages that *should* be delivered have been delivered. It is not an error if `Tracee` isn't, and hasn't been traced by someone, but if this is the case, *no* trace messages will have been delivered when the `trace_delivered` message arrives.

Note that `Tracee` has to refer to a process currently, or previously existing on the same node as the caller of `erlang:trace_delivered(Tracee)` resides on. The special `Tracee` atom `all` denotes all processes that currently are traced in the node.

An example: Process `A` is tracee, port `B` is tracer, and process `C` is the port owner of `B`. `C` wants to close `B` when `A` exits. `C` can ensure that the trace isn't truncated by calling `erlang:trace_delivered(A)` when `A` exits and wait for the `{trace_delivered, A, Ref}` message before closing `B`.

Failure: `badarg` if `Tracee` does not refer to a process (dead or alive) on the same node as the caller of `erlang:trace_delivered(Tracee)` resides on.

`erlang:trace_info(PidOrFunc, Item) -> Res`

Types:

PidOrFunc = pid() | new | {Module, Function, Arity} | on_load

Module = Function = atom()

Arity = int()

Item, Res -- see below

Returns trace information about a process or function.

To get information about a process, `PidOrFunc` should be a pid or the atom `new`. The atom `new` means that the default trace state for processes to be created will be returned. `Item` must have one of the following values:

`flags`

Return a list of atoms indicating what kind of traces is enabled for the process. The list will be empty if no traces are enabled, and one or more of the followings atoms if traces are enabled: `send`, `'receive'`, `set_on_spawn`, `call`, `return_to`, `procs`, `set_on_first_spawn`, `set_on_link`, `running`, `garbage_collection`, `timestamp`, and `arity`. The order is arbitrary.

`tracer`

Return the identifier for process or port tracing this process. If this process is not being traced, the return value will be `[]`.

To get information about a function, `PidOrFunc` should be a three-element tuple: `{Module, Function, Arity}` or the atom `on_load`. No wildcards are allowed. Returns `undefined` if the function does not exist or `false` if the function is not traced at all. `Item` must have one of the following values:

`traced`

Return `global` if this function is traced on global function calls, `local` if this function is traced on local function calls (i.e local and global function calls), and `false` if neither local nor global function calls are traced.

`match_spec`

Return the match specification for this function, if it has one. If the function is locally or globally traced but has no match specification defined, the returned value is `[]`.

`meta`

Return the meta trace tracer process or port for this function, if it has one. If the function is not meta traced the returned value is `false`, and if the function is meta traced but has once detected that the tracer proc is invalid, the returned value is [].

`meta_match_spec`

Return the meta trace match specification for this function, if it has one. If the function is meta traced but has no match specification defined, the returned value is `[]`.

`call_count`

Return the call count value for this function or `true` for the pseudo function `on_load` if call count tracing is active. Return `false` otherwise. See also *erlang:trace_pattern/3*.

`all`

Return a list containing the `{Item, Value}` tuples for all other items, or return `false` if no tracing is active for this function.

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

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

`erlang:trace_pattern(MFA, MatchSpec) -> int()`

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

erlang:trace_pattern(MFA, MatchSpec, FlagList) -> int()

Types:

MFA, MatchSpec, FlagList -- see below

This BIF is used to enable or disable call tracing for exported functions. It must be combined with *erlang:trace/3* to set the `call` trace flag for one or more processes.

Conceptually, call tracing works like this: Inside the Erlang virtual machine there is a set of processes to be traced and a set of functions to be traced. Tracing will be enabled on the intersection of the set. That is, if a process included in the traced process set calls a function included in the traced function set, the trace action will be taken. Otherwise, nothing will happen.

Use *erlang:trace/3* to add or remove one or more processes to the set of traced processes. Use `erlang:trace_pattern/2` to add or remove exported functions to the set of traced functions.

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

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

`{Module,Function,'_'}`

All exported functions of any arity named `Function` in module `Module`.

`{Module,'_','_'}`

All exported functions in module `Module`.

`{'_','_','_'}`

All exported functions in all loaded modules.

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

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

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

`false`

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

`true`

Enable tracing for the matching function(s).

`MatchSpecList`

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

`restart`

For the `FlagList` option `call_count`: restart the existing counters. The behaviour is undefined for other `FlagList` options.

`pause`

For the `FlagList` option `call_count`: pause the existing counters. The behaviour is undefined for other `FlagList` options.

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

`global`

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

`local`

Turn on or off call tracing for all types of function calls. Trace messages will be sent whenever any of the specified functions are called, regardless of how they are called. If the `return_to` flag is set for the process, a `return_to` message will also be sent when this function returns to its caller.

`meta | {meta, Pid}`

Turn on or off meta tracing for all types of function calls. Trace messages will be sent to the tracer process or port `Pid` whenever any of the specified functions are called, regardless of how they are called. If no `Pid` is specified, `self()` is used as a default tracer process.

Meta tracing traces all processes and does not care about the process trace flags set by `trace/3`, the trace flags are instead fixed to `[call, timestamp]`.

The match spec function `{return_trace}` works with meta trace and send its trace message to the same tracer process.

`call_count`

Starts (`MatchSpec == true`) or stops (`MatchSpec == false`) call count tracing for all types of function calls. For every function a counter is incremented when the function is called, in any process. No process trace flags need to be activated.

If call count tracing is started while already running, the count is restarted from zero. Running counters can be paused with `MatchSpec == pause`. Paused and running counters can be restarted from zero with `MatchSpec == restart`.

The counter value can be read with *erlang:trace_info/2*.

The `global` and `local` options are mutually exclusive and `global` is the default (if no options are specified). The `call_count` and `meta` options perform a kind of local tracing, and can also not be combined with `global`. A function can be either globally or locally traced. If global tracing is specified for a specified set of functions; local, meta and call count tracing for the matching set of local functions will be disabled, and vice versa.

When disabling trace, the option must match the type of trace that is set on the function, so that local tracing must be disabled with the `local` option and global tracing with the `global` option (or no option at all), and so forth.

There is no way to directly change part of a match specification list. If a function has a match specification, you can replace it with a completely new one. If you need to change an existing match specification, use the *erlang:trace_info/2* BIF to retrieve the existing match specification.

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

`trunc(Number) -> int()`

Types:

Number = number()

Returns an integer by the truncating `Number`.

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

Allowed in guard tests.

tuple_size(Tuple) -> int()

Types:

Tuple = tuple()

Returns an integer which is the number of elements in `Tuple`.

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

Allowed in guard tests.

tuple_to_list(Tuple) -> [term()]

Types:

Tuple = tuple()

Returns a list which corresponds to `Tuple`. `Tuple` may contain any Erlang terms.

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

erlang:universaltime() -> {Date, Time}

Types:

Date = {Year, Month, Day}

Time = {Hour, Minute, Second}

Year = Month = Day = Hour = Minute = Second = int()

Returns the current date and time according to Universal Time Coordinated (UTC), also called GMT, in the form `{{Year, Month, Day}, {Hour, Minute, Second}}` if supported by the underlying operating system. If not, `erlang:universaltime()` is equivalent to `erlang:localtime()`.

```
> erlang:universaltime().
{{1996,11,6},{14,18,43}}
```

erlang:universaltime_to_localtime({Date1, Time1}) -> {Date2, Time2}

Types:

Date1 = Date2 = {Year, Month, Day}

Time1 = Time2 = {Hour, Minute, Second}

Year = Month = Day = Hour = Minute = Second = int()

Converts Universal Time Coordinated (UTC) date and time to local date and time, if this is supported by the underlying OS. Otherwise, no conversion is done, and `{Date1, Time1}` is returned.

```
> erlang:universaltime_to_localtime({{1996,11,6},{14,18,43}}).
{{1996,11,7},{15,18,43}}
```

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

`unlink(Id) -> true`

Types:

Id = pid() | port()

Removes the link, if there is one, between the calling process and the process or port referred to by `Id`.

Returns `true` and does not fail, even if there is no link to `Id`, or if `Id` does not exist.

Once `unlink(Id)` has returned it is guaranteed that the link between the caller and the entity referred to by `Id` has no effect on the caller in the future (unless the link is setup again). If caller is trapping exits, an `{'EXIT', Id, _}` message due to the link might have been placed in the callers message queue prior to the call, though. Note, the `{'EXIT', Id, _}` message can be the result of the link, but can also be the result of `Id` calling `exit/2`. Therefore, it *may* be appropriate to cleanup the message queue when trapping exits after the call to `unlink(Id)`, as follow:

```
unlink(Id),
receive
    {'EXIT', Id, _} ->
        true
after 0 ->
        true
end
```

> Note:
>
> Prior to OTP release R11B (erts version 5.5) `unlink/1` behaved completely asynchronous, i.e., the link was active until the "unlink signal" reached the linked entity. This had one undesirable effect, though. You could never know when you were guaranteed *not* to be effected by the link.
>
> Current behavior can be viewed as two combined operations: asynchronously send an "unlink signal" to the linked entity and ignore any future results of the link.

`unregister(RegName) -> true`

Types:

RegName = atom()

Removes the registered name `RegName`, associated with a pid or a port identifier.

```
> unregister(db).
true
```

Users are advised not to unregister system processes.

Failure: `badarg` if `RegName` is not a registered name.

`whereis(RegName) -> pid() | port() | undefined`

Returns the pid or port identifier with the registered name `RegName`. Returns `undefined` if the name is not registered.

```
> whereis(db).
<0.43.0>
```

`erlang:yield() -> true`

Voluntarily let other processes (if any) get a chance to execute. Using `erlang:yield()` is similar to `receive after 1 -> ok end`, except that `yield()` is faster.

> Warning:
>
> There is seldom or never any need to use this BIF, especially in the SMP-emulator as other processes will have a chance to run in another scheduler thread anyway. Using this BIF without a thorough grasp of how the scheduler works may cause performance degradation.

init

Erlang module

The `init` module is pre-loaded and contains the code for the `init` system process which coordinates the start-up of the system. The first function evaluated at start-up is `boot(BootArgs)`, where `BootArgs` is a list of command line arguments supplied to the Erlang runtime system from the local operating system. See *erl(1)*.

`init` reads the boot script which contains instructions on how to initiate the system. See *script(4)* for more information about boot scripts.

`init` also contains functions to restart, reboot, and stop the system.

Exports

`boot(BootArgs) -> void()`

Types:

BootArgs = [binary()]

Starts the Erlang runtime system. This function is called when the emulator is started and coordinates system start-up.

`BootArgs` are all command line arguments except the emulator flags, that is, flags and plain arguments. See *erl(1)*.

`init` itself interprets some of the flags, see *Command Line Flags* below. The remaining flags ("user flags") and plain arguments are passed to the `init` loop and can be retrieved by calling `get_arguments/0` and `get_plain_arguments/0`, respectively.

`get_args() -> [Arg]`

Types:

Arg = atom()

Returns any plain command line arguments as a list of atoms (possibly empty). It is recommended that `get_plain_arguments/1` is used instead, because of the limited length of atoms.

`get_argument(Flag) -> {ok, Arg} | error`

Types:

Flag = atom()

Arg = [Values]

Values = [string()]

Returns all values associated with the command line user flag `Flag`. If `Flag` is provided several times, each `Values` is returned in preserved order.

```
% erl -a b c -a d
...
1> init:get_argument(a).
{ok,[["b","c"],["d"]]}
```

There are also a number of flags, which are defined automatically and can be retrieved using this function:

`root`

The installation directory of Erlang/OTP, `$ROOT`.

```
2> init:get_argument(root).
{ok,[["/usr/local/otp/releases/otp_beam_solaris8_r10b_patched"]]}
```

`progname`

The name of the program which started Erlang.

```
3> init:get_argument(progname).
{ok,[["erl"]]}
```

`home`

The home directory.

```
4> init:get_argument(home).
{ok,[["/home/harry"]]}
```

Returns `error` if there is no value associated with `Flag`.

`get_arguments() -> Flags`

Types:

Flags = [{Flag, Values}]

Flag = atom()

Values = [string()]

Returns all command line flags, as well as the system defined flags, see `get_argument/1`.

`get_plain_arguments() -> [Arg]`

Types:

Arg = string()

Returns any plain command line arguments as a list of strings (possibly empty).

`get_status() -> {InternalStatus, ProvidedStatus}`

Types:

InternalStatus = starting | started | stopping

ProvidedStatus = term()

The current status of the `init` process can be inspected. During system startup (initialization), `InternalStatus` is `starting`, and `ProvidedStatus` indicates how far the boot script has been interpreted. Each `{progress, Info}` term interpreted in the boot script affects `ProvidedStatus`, that is, `ProvidedStatus` gets the value of `Info`.

`reboot() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` command line flag was given, the `heart` program will try to reboot the system. Refer to `heart(3)` for more information.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`restart() -> void()`

The system is restarted *inside* the running Erlang node, which means that the emulator is not restarted. All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system is booted again in the same way as initially started. The same `BootArgs` are used again.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`script_id() -> Id`

Types:

Id = term()

Get the identity of the boot script used to boot the system. `Id` can be any Erlang term. In the delivered boot scripts, `Id` is `{Name, Vsn}`. `Name` and `Vsn` are strings.

`stop() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` command line flag was given, the `heart` program is terminated before the Erlang node terminates. Refer to `heart(3)` for more information.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`stop(Status) -> void()`

Types:

Status = int()>=0 | string()

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates by calling `halt(Status)`. If the `-heart` command line flag was given, the `heart` program is terminated before the Erlang node terminates. Refer to `heart(3)` for more information.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

Command Line Flags

> ### Warning:
>
> The support for loading of code from archive files is experimental. The sole purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces etc. may be changed in a future release. The `-code_path_choice` flag is also experimental.

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

`--`

Everything following `--` up to the next flag is considered plain arguments and can be retrieved using `get_plain_arguments/0`.

`-code_path_choice Choice`

This flag can be set to `strict` or `relaxed`. It controls whether each directory in the code path should be interpreted strictly as it appears in the `boot script` or if `init` should be more relaxed and try to find a suitable directory if it can choose from a regular ebin directory and an ebin directory in an archive file. This flag is particular useful when you want to elaborate with code loading from archives without editing the `boot script`. See *script(4)* for more information about interpretation of boot scripts. The flag does also have a similar affect on how the code server works. See *code(3)*.

`-eval Expr`

Scans, parses and evaluates an arbitrary expression `Expr` during system initialization. If any of these steps fail (syntax error, parse error or exception during evaluation), Erlang stops with an error message. Here is an example that seeds the random number generator:

```
% erl -eval '{X,Y,Z}' = now(), random:seed(X,Y,Z).'
```

This example uses Erlang as a hexadecimal calculator:

```
% erl -noshell -eval 'R = 16#1F+16#A0, io:format("~.16B~n", [R])' \\
-s erlang halt
BF
```

If multiple `-eval` expressions are specified, they are evaluated sequentially in the order specified. `-eval` expressions are evaluated sequentially with `-s` and `-run` function calls (this also in the order specified). As with `-s` and `-run`, an evaluation that does not terminate, blocks the system initialization process.

`-extra`

Everything following `-extra` is considered plain arguments and can be retrieved using `get_plain_arguments/0`.

`-run Mod [Func [Arg1, Arg2, ...]]`

Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as strings. If an exception is raised, Erlang stops with an error message.

Example:

```
% erl -run foo -run foo bar -run foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```
foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `-run` call which does not return will block further processing; to avoid this, use some variant of `spawn` in such cases.

`-s Mod [Func [Arg1, Arg2, ...]]`

Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as atoms. If an exception is raised, Erlang stops with an error message.

Example:

```
% erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `-s` call which does not return will block further processing; to avoid this, use some variant of `spawn` in such cases.

Due to the limited length of atoms, it is recommended that `-run` be used instead.

Example

```
% erl -- a b -children thomas claire -ages 7 3 -- x y
...

1> init:get_plain_arguments().
["a","b","x","y"]
2> init:get_argument(children).
{ok,[["thomas","claire"]]}
3> init:get_argument(ages).
{ok, [["7","3"]]}
4> init:get_argument(silly).
error
```

SEE ALSO

erl_prim_loader(3), heart(3)

zlib

Erlang module

The zlib module provides an API for the zlib library (http://www.zlib.org). It is used to compress and decompress data. The data format is described by RFCs 1950 to 1952.

A typical (compress) usage looks like:

```
Z = zlib:open(),
ok = zlib:deflateInit(Z,default),

Compress = fun(end_of_data, _Cont) -> [];
              (Data, Cont) ->
                 [zlib:deflate(Z, Data)|Cont(Read(),Cont)]
           end,
Compressed = Compress(Read(),Compress),
Last = zlib:deflate(Z, [], finish),
ok = zlib:deflateEnd(Z),
zlib:close(Z),
list_to_binary([Compressed|Last])
```

In all functions errors, `{'EXIT',{Reason,Backtrace}}`, might be thrown, where `Reason` describes the error. Typical reasons are:

`badarg`

Bad argument

`data_error`

The data contains errors

`stream_error`

Inconsistent stream state

`einval`

Bad value or wrong function called

`{need_dictionary,Adler32}`

See `inflate/2`

DATA TYPES

```
iodata = iolist() | binary()

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

zstream = a zlib stream, see open/0
```

Exports

`open() -> Z`

Types:

Z = zstream()

Open a zlib stream.

`close(Z) -> ok`

Types:

Z = zstream()

Closes the stream referenced by `Z`.

`deflateInit(Z) -> ok`

Types:

Z = zstream()

Same as `zlib:deflateInit(Z, default)`.

`deflateInit(Z, Level) -> ok`

Types:

Z = zstream()

Level = none | default | best_speed | best_compression | 0..9

Initialize a zlib stream for compression.

`Level` decides the compression level to be used, 0 (`none`), gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression.

`deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok`

Types:

Z = zstream()

Level = none | default | best_speed | best_compression | 0..9

Method = deflated

WindowBits = 9..15|-9..-15

MemLevel = 1..9

Strategy = default|filtered|huffman_only

Initiates a zlib stream for compression.

The `Level` parameter decides the compression level to be used, 0 (`none`), gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression.

The `Method` parameter decides which compression method to use, currently the only supported method is `deflated`.

The `WindowBits` parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 9 through 15. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if `deflateInit/2`. A negative `WindowBits` value suppresses the zlib header (and checksum) from the stream. Note that the zlib source mentions this only as a undocumented feature.

The `MemLevel` parameter specifies how much memory should be allocated for the internal compression state. `MemLevel`=1 uses minimum memory but is slow and reduces compression ratio; `MemLevel`=9 uses maximum memory for optimal speed. The default value is 8.

The `Strategy` parameter is used to tune the compression algorithm. Use the value `default` for normal data, `filtered` for data produced by a filter (or predictor), or `huffman_only` to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of `filtered`is to force more Huffman coding and less string matching; it is somewhat intermediate between `default` and `huffman_only`. The `Strategy` parameter only affects the compression ratio but not the correctness of the compressed output even if it is not set appropriately.

`deflate(Z, Data) -> Compressed`

Types:

Z = zstream()

Data = iodata()

Compressed = iolist()

Same as `deflate(Z, Data, none)`.

`deflate(Z, Data, Flush) ->`

Types:

Z = zstream()

Data = iodata()

Flush = none | sync | full | finish

Compressed = iolist()

`deflate/3` compresses as much data as possible, and stops when the input buffer becomes empty. It may introduce some output latency (reading input without producing any output) except when forced to flush.

If the parameter `Flush` is set to `sync`, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.

If `Flush` is set to `full`, all output is flushed as with `sync`, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using `full` too often can seriously degrade the compression.

If the parameter `Flush` is set to `finish`, pending input is processed, pending output is flushed and `deflate/3` returns. Afterwards the only possible operations on the stream are `deflateReset/1` or `deflateEnd/1`.

`Flush` can be set to `finish` immediately after `deflateInit` if all compression is to be done in one step.

```
zlib:deflateInit(Z),
B1 = zlib:deflate(Z,Data),
B2 = zlib:deflate(Z,<< >>,finish),
zlib:deflateEnd(Z),
list_to_binary([B1,B2])
```

`deflateSetDictionary(Z, Dictionary) -> Adler32`

Types:

Z = zstream()

Dictionary = binary()

Adler32 = integer()

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after `deflateInit/[1|2|6]` or `deflateReset/1`, before any call of `deflate/3`. The compressor and decompressor must use exactly the same dictionary (see `inflateSetDictionary/2`). The adler checksum of the dictionary is returned.

`deflateReset(Z) -> ok`

Types:

Z = zstream()

This function is equivalent to `deflateEnd/1` followed by `deflateInit/[1|2|6]`, but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes.

`deflateParams(Z, Level, Strategy) -> ok`

Types:

Z = zstream()

Level = none | default | best_speed | best_compression | 0..9

Strategy = default|filtered|huffman_only

Dynamically update the compression level and compression strategy. The interpretation of `Level` and `Strategy` is as in `deflateInit/6`. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call of `deflate/3`.

Before the call of deflateParams, the stream state must be set as for a call of `deflate/3`, since the currently available input may have to be compressed and flushed.

`deflateEnd(Z) -> ok`

Types:

Z = zstream()

End the deflate session and cleans all data used. Note that this function will throw an `data_error` exception if the last call to `deflate/3` was not called with `Flush` set to `finish`.

`inflateInit(Z) -> ok`

Types:

Z = zstream()

Initialize a zlib stream for decompression.

`inflateInit(Z, WindowBits) -> ok`

Types:

Z = zstream()

WindowBits = 9..15|-9..-15

Initialize decompression session on zlib stream.

The `WindowBits` parameter is the base two logarithm of the maximum window size (the size of the history buffer). It should be in the range 9 through 15. The default value is 15 if `inflateInit/1` is used. If a compressed stream with a larger window size is given as input, inflate() will throw the `data_error` exception. A negative `WindowBits` value makes zlib ignore the zlib header (and checksum) from the stream. Note that the zlib source mentions this only as a undocumented feature.

`inflate(Z, Data) -> DeCompressed`

Types:

Z = zstream()

Data = iodata()

DeCompressed = iolist()

`inflate/2` decompresses as much data as possible. It may some introduce some output latency (reading input without producing any output).

If a preset dictionary is needed at this point (see `inflateSetDictionary` below), `inflate/2` throws a `{need_dictionary,Adler}` exception where `Adler` is the adler32 checksum of the dictionary chosen by the compressor.

`inflateSetDictionary(Z, Dictionary) -> ok`

Types:

Z = zstream()

Dictionary = binary()

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call of `inflate/2` if this call threw a `{need_dictionary,Adler}` exception. The dictionary chosen by the compressor can be determined from the Adler value thrown by the call to `inflate/2`. The compressor and decompressor must use exactly the same dictionary (see `deflateSetDictionary/2`).

Example:

```
unpack(Z, Compressed, Dict) ->
     case catch zlib:inflate(Z, Compressed) of
          {'EXIT',{{need_dictionary,DictID},_}} ->
                   zlib:inflateSetDictionary(Z, Dict),
                 Uncompressed = zlib:inflate(Z, []);
          Uncompressed ->
                 Uncompressed
     end.
```

`inflateReset(Z) -> ok`

Types:

Z = zstream()

This function is equivalent to `inflateEnd/1` followed by `inflateInit/1`, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by `inflateInit/[1|2]`.

`inflateEnd(Z) -> ok`

Types:

Z = zstream()

End the inflate session and cleans all data used. Note that this function will throw a `data_error` exception if no end of stream was found (meaning that not all data has been uncompressed).

`setBufSize(Z, Size) -> ok`

Types:

Z = zstream()

Size = integer()

Sets the intermediate buffer size.

`getBufSize(Z) -> Size`

Types:

Z = zstream()

Size = integer()

Get the size of intermediate buffer.

`crc32(Z) -> CRC`

Types:

Z = zstream()

CRC = integer()

Get the current calculated CRC checksum.

`crc32(Z, Binary) -> CRC`

Types:

Z = zstream()

Binary = binary()

CRC = integer()

Calculate the CRC checksum for `Binary`.

`crc32(Z, PrevCRC, Binary) -> CRC`

Types:

Z = zstream()

PrevCRC = integer()

Binary = binary()

CRC = integer()

Update a running CRC checksum for `Binary`. If `Binary` is the empty binary, this function returns the required initial value for the crc.

```
Crc = lists:foldl(fun(Bin,Crc0) ->
                      zlib:crc32(Z, Crc0, Bin),
                  end, zlib:crc32(Z,<< >>), Bins)
```

`crc32_combine(Z, CRC1, CRC2, Size2) -> CRC`

Types:

Z = zstream()

CRC = integer()

CRC1 = integer()

CRC2 = integer()

Size2 = integer()

Combine two CRC checksums into one. For two binaries, `Bin1` and `Bin2` with sizes of `Size1` and `Size2`, with CRC checksums `CRC1` and `CRC2`. `crc32_combine/4` returns the `CRC` checksum of `<<Bin1/binary,Bin2/binary>>`, requiring only `CRC1`, `CRC2`, and `Size2`.

adler32(Z, Binary) -> Checksum

Types:

Z = zstream()

Binary = binary()

Checksum = integer()

Calculate the Adler-32 checksum for `Binary`.

adler32(Z, PrevAdler, Binary) -> Checksum

Types:

Z = zstream()

PrevAdler = integer()

Binary = binary()

Checksum = integer()

Update a running Adler-32 checksum for `Binary`. If `Binary` is the empty binary, this function returns the required initial value for the checksum.

```
Crc = lists:foldl(fun(Bin,Crc0) ->
                      zlib:adler32(Z, Crc0, Bin),
                  end, zlib:adler32(Z,<< >>), Bins)
```

adler32_combine(Z, Adler1, Adler2, Size2) -> Adler

Types:

Z = zstream()

Adler = integer()

Adler1 = integer()

Adler2 = integer()

Size2 = integer()

Combine two Adler-32 checksums into one. For two binaries, `Bin1` and `Bin2` with sizes of `Size1` and `Size2`, with Adler-32 checksums `Adler1` and `Adler2`. `adler32_combine/4` returns the `Adler` checksum of `<<Bin1/binary,Bin2/binary>>`, requiring only `Adler1`, `Adler2`, and `Size2`.

compress(Binary) -> Compressed

Types:

Binary = Compressed = binary()

Compress a binary (with zlib headers and checksum).

`uncompress(Binary) -> Decompressed`

Types:

Binary = Decompressed = binary()

Uncompress a binary (with zlib headers and checksum).

`zip(Binary) -> Compressed`

Types:

Binary = Compressed = binary()

Compress a binary (without zlib headers and checksum).

`unzip(Binary) -> Decompressed`

Types:

Binary = Decompressed = binary()

Uncompress a binary (without zlib headers and checksum).

`gzip(Data) -> Compressed`

Types:

Binary = Compressed = binary()

Compress a binary (with gz headers and checksum).

`gunzip(Bin) -> Decompressed`

Types:

Binary = Decompressed = binary()

Uncompress a binary (with gz headers and checksum).

epmd

Command

This daemon acts as a name server on all hosts involved in distributed Erlang computations. When an Erlang node starts, the node has a name and it obtains an address from the host OS kernel. The name and the address are sent to the `epmd` daemon running on the local host. In a TCP/IP environment, the address consists of the IP address and a port number. The name of the node is an atom on the form of `Name@Node`. The job of the `epmd` daemon is to keep track of which node name listens on which address. Hence, `epmd` map symbolic node names to machine addresses.

The daemon is started automatically by the Erlang start-up script.

The program `epmd` can also be used for a variety of other purposes, for example checking the DNS (Domain Name System) configuration of a host.

Exports

epmd [-daemon]

Starts a name server as a daemon. If it has no argument, the `epmd` runs as a normal program with the controlling terminal of the shell in which it is started. Normally, it should run as a daemon.

epmd -names

Requests the names of the local Erlang nodes `epmd` has registered.

epmd -kill

Kills the `epmd` process.

epmd -help

Write short info about the usage including some debugging options not listed here.

Environment variables

`ERL_EPMD_PORT`

> This environment variable can contain the port number epmd will use. The default port will work fine in most cases. A different port can be specified to allow several instances of epmd, representing independent clusters of nodes, to co-exist on the same host. All nodes in a cluster must use the same epmd port number.

Logging

On some operating systems *syslog* will be used for error reporting when epmd runs as an daemon. To enable the error logging you have to edit /etc/syslog.conf file and add an entry

```
!epmd
*.*<TABs>/var/log/epmd.log
```

where <TABs> are at least one real tab character. Spaces will silently be ignored.

erl

Command

The `erl` program starts an Erlang runtime system. The exact details (for example, whether `erl` is a script or a program and which other programs it calls) are system-dependent.

Windows users probably wants to use the `werl` program instead, which runs in its own window with scrollbars and supports command-line editing. The `erl` program on Windows provides no line editing in its shell, and on Windows 95 there is no way to scroll back to text which has scrolled off the screen. The `erl` program must be used, however, in pipelines or if you want to redirect standard input or output.

Exports

erl <arguments>

Starts an Erlang runtime system.

The arguments can be divided into *emulator flags*, *flags* and *plain arguments*:

- Any argument starting with the character `+` is interpreted as an *emulator flag*.

  As indicated by the name, emulator flags controls the behavior of the emulator.
- Any argument starting with the character `-` (hyphen) is interpreted as a *flag* which should be passed to the Erlang part of the runtime system, more specifically to the `init` system process, see *init(3)*.

  The `init` process itself interprets some of these flags, the *init flags*. It also stores any remaining flags, the *user flags*. The latter can be retrieved by calling `init:get_argument/1`.

  It can be noted that there are a small number of "-" flags which now actually are emulator flags, see the description below.
- Plain arguments are not interpreted in any way. They are also stored by the `init` process and can be retrieved by calling `init:get_plain_arguments/0`. Plain arguments can occur before the first flag, or after a `--` flag. Additionally, the flag `-extra` causes everything that follows to become plain arguments.

Example:

```
% erl +W w -sname arnie +R 9 -s my_init -extra +bertie
(arnie@host)1> init:get_argument(sname).
{ok,[["arnie"]]}
(arnie@host)2> init:get_plain_arguments().
["+bertie"]
```

Here `+W w` and `+R 9` are emulator flags. `-s my_init` is an init flag, interpreted by `init`. `-sname arnie` is a user flag, stored by `init`. It is read by Kernel and will cause the Erlang runtime system to become distributed. Finally, everything after `-extra` (that is, `+bertie`) is considered as plain arguments.

```
% erl -myflag 1
1> init:get_argument(myflag).
{ok,[["1"]]}
2> init:get_plain_arguments().
[]
```

Here the user flag `-myflag 1` is passed to and stored by the `init` process. It is a user defined flag, presumably used by some user defined application.

Flags

In the following list, init flags are marked (init flag). Unless otherwise specified, all other flags are user flags, for which the values can be retrieved by calling `init:get_argument/1`. Note that the list of user flags is not exhaustive, there may be additional, application specific flags which instead are documented in the corresponding application documentation.

`--`(init flag)

Everything following `--` up to the next flag (`-flag` or `+flag`) is considered plain arguments and can be retrieved using `init:get_plain_arguments/0`.

`-Application Par Val`

Sets the application configuration parameter `Par` to the value `Val` for the application `Application`, see *app(4)* and *application(3)*.

`-args_file FileName`

Command line arguments are read from the file `FileName`. The arguments read from the file replace the '`-args_file FileName`' flag on the resulting command line.

The file `FileName` should be a plain text file and may contain comments and command line arguments. A comment begins with a # character and continues until next end of line character. Backslash (\\) is used as quoting character. All command line arguments accepted by `erl` are allowed, also the `-args_file FileName` flag. Be careful not to cause circular dependencies between files containing the `-args_file` flag, though.

The `-extra` flag is treated specially. Its scope ends at the end of the file. Arguments following an `-extra` flag are moved on the command line into the `-extra` section, i.e. the end of the command line following after an `-extra` flag.

`-async_shell_start`

The initial Erlang shell does not read user input until the system boot procedure has been completed (Erlang 5.4 and later). This flag disables the start synchronization feature and lets the shell start in parallel with the rest of the system.

`-boot File`

Specifies the name of the boot file, `File.boot`, which is used to start the system. See *init(3)*. Unless `File` contains an absolute path, the system searches for `File.boot` in the current and `$ROOT/bin` directories.

Defaults to `$ROOT/bin/start.boot`.

`-boot_var Var Dir`

If the boot script contains a path variable `Var` other than `$ROOT`, this variable is expanded to `Dir`. Used when applications are installed in another directory than `$ROOT/lib`, see *systools:make_script/1,2*.

`-code_path_cache`

Enables the code path cache of the code server, see *code(3)*.

`-compile Mod1 Mod2 ...`

Compiles the specified modules and then terminates (with non-zero exit code if the compilation of some file did not succeed). Implies `-noinput`. Not recommended - use *erlc* instead.

`-config Config`

Specifies the name of a configuration file, `Config.config`, which is used to configure applications. See *app(4)* and *application(3)*.

`-connect_all false`

If this flag is present, `global` will not maintain a fully connected network of distributed Erlang nodes, and then global name registration cannot be used. See *global(3)*.

`-cookie Cookie`

Obsolete flag without any effect and common misspelling for `-setcookie`. Use `-setcookie` instead.

`-detached`

Starts the Erlang runtime system detached from the system console. Useful for running daemons and backgrounds processes.

`-emu_args`

Useful for debugging. Prints out the actual arguments sent to the emulator.

`-env Variable Value`

Sets the host OS environment variable `Variable` to the value `Value` for the Erlang runtime system. Example:

```
% erl -env DISPLAY gin:0
```

In this example, an Erlang runtime system is started with the `DISPLAY` environment variable set to `gin:0`.

`-eval Expr`(init flag)

Makes `init` evaluate the expression `Expr`, see *init(3)*.

`-extra`(init flag)

Everything following `-extra` is considered plain arguments and can be retrieved using `init:get_plain_arguments/0`.

`-heart`

Starts heart beat monitoring of the Erlang runtime system. See *heart(3)*.

`-hidden`

Starts the Erlang runtime system as a hidden node, if it is run as a distributed node. Hidden nodes always establish hidden connections to all other nodes except for nodes in the same global group. Hidden connections are not published on either of the connected nodes, i.e. neither of the connected nodes are part of the result from `nodes/0` on the other node. See also hidden global groups, *global_group(3)*.

`-hosts Hosts`

Specifies the IP addresses for the hosts on which Erlang boot servers are running, see *erl_boot_server(3)*. This flag is mandatory if the `-loader inet` flag is present.

The IP addresses must be given in the standard form (four decimal numbers separated by periods, for example `"150.236.20.74"`. Hosts names are not acceptable, but a broadcast address (preferably limited to the local network) is.

`-id Id`

Specifies the identity of the Erlang runtime system. If it is run as a distributed node, `Id` must be identical to the name supplied together with the `-sname` or `-name` flag.

`-init_debug`

Makes `init` write some debug information while interpreting the boot script.

`-instr`(emulator flag)

Selects an instrumented Erlang runtime system (virtual machine) to run, instead of the ordinary one. When running an instrumented runtime system, some resource usage data can be obtained and analysed using the module `instrument`. Functionally, it behaves exactly like an ordinary Erlang runtime system.

`-loader Loader`

Specifies the method used by `erl_prim_loader` to load Erlang modules into the system. See *erl_prim_loader(3)*. Two `Loader` methods are supported, `efile` and `inet`. `efile` means use the local file system, this is the default. `inet` means use a boot server on another machine, and the `-id`, `-hosts` and `-setcookie` flags must be specified as well. If `Loader` is something else, the user supplied `Loader` port program is started.

`-make`

Makes the Erlang runtime system invoke `make:all()` in the current working directory and then terminate. See *make(3)*. Implies `-noinput`.

`-man Module`

Displays the manual page for the Erlang module `Module`. Only supported on Unix.

`-mode interactive | embedded`

Indicates if the system should load code dynamically (`interactive`), or if all code should be loaded during system initialization (`embedded`), see *code(3)*. Defaults to `interactive`.

`-name Name`

Makes the Erlang runtime system into a distributed node. This flag invokes all network servers necessary for a node to become distributed. See *net_kernel(3)*. It is also ensured that `epmd` runs on the current host before Erlang is started. See *epmd(1)*.

The name of the node will be `Name@Host`, where `Host` is the fully qualified host name of the current host. For short names, use the `-sname` flag instead.

`-noinput`

Ensures that the Erlang runtime system never tries to read any input. Implies `-noshell`.

`-noshell`

Starts an Erlang runtime system with no shell. This flag makes it possible to have the Erlang runtime system as a component in a series of UNIX pipes.

`-nostick`

Disables the sticky directory facility of the Erlang code server, see *code(3)*.

`-oldshell`

Invokes the old Erlang shell from Erlang 3.3. The old shell can still be used.

`-pa Dir1 Dir2 ...`

Adds the specified directories to the beginning of the code path, similar to `code:add_pathsa/1`. See *code(3)*. As an alternative to `-pa`, if several directories are to be prepended to the code and the directories have a common parent directory, that parent directory could be specified in the `ERL_LIBS` environment variable. See *code(3)*.

`-pz Dir1 Dir2 ...`

Adds the specified directories to the end of the code path, similar to `code:add_pathsz/1`. See *code(3)*.

`-remsh Node`

Starts Erlang with a remote shell connected to `Node`.

`-rsh Program`

Specifies an alternative to `rsh` for starting a slave node on a remote host. See *slave(3)*.

`-run Mod [Func [Arg1, Arg2, ...]]`(init flag)

Makes `init` call the specified function. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as strings. See *init(3)*.

`-s Mod [Func [Arg1, Arg2, ...]]`(init flag)

Makes `init` call the specified function. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as atoms. See *init(3)*.

`-setcookie Cookie`

Sets the magic cookie of the node to `Cookie`, see *erlang:set_cookie/2*.

`-shutdown_time Time`

Specifies how long time (in milliseconds) the `init` process is allowed to spend shutting down the system. If `Time` ms have elapsed, all processes still existing are killed. Defaults to `infinity`.

`-sname Name`

Makes the Erlang runtime system into a distributed node, similar to `-name`, but the host name portion of the node name `Name@Host` will be the short name, not fully qualified.

This is sometimes the only way to run distributed Erlang if the DNS (Domain Name System) is not running. There can be no communication between nodes running with the `-sname` flag and those running with the `-name` flag, as node names must be unique in distributed Erlang systems.

`-smp [enable|auto|disable]`

`-smp enable` and `-smp` starts the Erlang runtime system with SMP support enabled. This may fail if no runtime system with SMP support is available. `-smp auto` starts the Erlang runtime system with SMP support enabled if it is available and more than one logical processor are detected. `-smp disable` starts a runtime system without SMP support. By default `-smp auto` will be used unless a conflicting parameter has been passed, then `-smp disable` will be used. Currently only the `-hybrid` parameter conflicts with `-smp auto`.

NOTE: The runtime system with SMP support will not be available on all supported platforms. See also the *+S* flag.

`-version`(emulator flag)

Makes the emulator print out its version number. The same as `erl +V`.

Emulator Flags

`erl` invokes the code for the Erlang emulator (virtual machine), which supports the following flags:

`+a size`

Suggested stack size, in kilowords, for threads in the async-thread pool. Valid range is 16-8192 kilowords. The default suggested stack size is 16 kilowords, i.e, 64 kilobyte on 32-bit architectures. This small default size has been chosen since the amount of async-threads might be quite large. The default size is enough for drivers delivered with Erlang/OTP, but might not be sufficiently large for other dynamically linked in drivers that use the *driver_async()* functionality. Note that the value passed is only a suggestion, and it might even be ignored on some platforms.

`+A size`

Sets the number of threads in async thread pool, valid range is 0-1024. Default is 0.

`+B [c | d | i]`

The `c` option makes `Ctrl-C` interrupt the current shell instead of invoking the emulator break handler. The `d` option (same as specifying `+B` without an extra option) disables the break handler. The `i` option makes the emulator ignore any break signal.

If the `c` option is used with `oldshell` on Unix, `Ctrl-C` will restart the shell process rather than interrupt it.

Note that on Windows, this flag is only applicable for `werl`, not `erl` (`oldshell`). Note also that `Ctrl-Break` is used instead of `Ctrl-C` on Windows.

`+c`

Disable compensation for sudden changes of system time.

Normally, `erlang:now/0` will not immediately reflect sudden changes in the system time, in order to keep timers (including `receive-after`) working. Instead, the time maintained by `erlang:now/0` is slowly adjusted towards the new system time. (Slowly means in one percent adjustments; if the time is off by one minute, the time will be adjusted in 100 minutes.)

When the `+c` option is given, this slow adjustment will not take place. Instead `erlang:now/0` will always reflect the current system time. Note that timers are based on `erlang:now/0`. If the system time jumps, timers then time out at the wrong time.

`+d`

If the emulator detects an internal error (or runs out of memory), it will by default generate both a crash dump and a core dump. The core dump will, however, not be very useful since the content of process heaps is destroyed by the crash dump generation.

The `+d` option instructs the emulator to only produce a core dump and no crash dump if an internal error is detected.

Calling `erlang:halt/1` with a string argument will still produce a crash dump.

`+hms Size`

Sets the default heap size of processes to the size `Size`.

`+hmbs Size`

Sets the default binary virtual heap size of processes to the size `Size`.

`+K true | false`

Enables or disables the kernel poll functionality if the emulator supports it. Default is `false` (disabled). If the emulator does not support kernel poll, and the `+K` flag is passed to the emulator, a warning is issued at startup.

`+l`

Enables auto load tracing, displaying info while loading code.

`+MFlag Value`

Memory allocator specific flags, see *erts_alloc(3)* for further information.

`+P Number`

Sets the maximum number of concurrent processes for this system. `Number` must be in the range 16..134217727. Default is 32768.

`+R ReleaseNumber`

Sets the compatibility mode.

The distribution mechanism is not backwards compatible by default. This flags sets the emulator in compatibility mode with an earlier Erlang/OTP release `ReleaseNumber`. The release number must be in the range

`7..<current release>`. This limits the emulator, making it possible for it to communicate with Erlang nodes (as well as C- and Java nodes) running that earlier release.

For example, an R10 node is not automatically compatible with an R9 node, but R10 nodes started with the `+R 9` flag can co-exist with R9 nodes in the same distributed Erlang system, they are R9-compatible.

Note: Make sure all nodes (Erlang-, C-, and Java nodes) of a distributed Erlang system is of the same Erlang/OTP release, or from two different Erlang/OTP releases X and Y, where *all* Y nodes have compatibility mode X.

For example: A distributed Erlang system can consist of R10 nodes, or of R9 nodes and R9-compatible R10 nodes, but not of R9 nodes, R9-compatible R10 nodes and "regular" R10 nodes, as R9 and "regular" R10 nodes are not compatible.

`+r`

Force ets memory block to be moved on realloc.

`+S Schedulers:SchedulerOnline`

Sets the amount of scheduler threads to create and scheduler threads to set online when SMP support has been enabled. Valid range for both values are 1-1024. If the Erlang runtime system is able to determine the amount of logical processors configured and logical processors available, `Schedulers` will default to logical processors configured, and `SchedulersOnline` will default to logical processors available; otherwise, the default values will be 1. `Schedulers` may be omitted if `:SchedulerOnline` is not and vice versa. The amount of schedulers online can be changed at run time via *erlang:system_flag(schedulers_online, SchedulersOnline)*.

This flag will be ignored if the emulator doesn't have SMP support enabled (see the *-smp* flag).

`+sFlag Value`

Scheduling specific flags.

`+sbt BindType`

Set scheduler bind type. Currently valid `BindType`s:

`u`

Same as *erlang:system_flag(scheduler_bind_type, unbound)*.

`ns`

Same as *erlang:system_flag(scheduler_bind_type, no_spread)*.

`ts`

Same as *erlang:system_flag(scheduler_bind_type, thread_spread)*.

`ps`

Same as *erlang:system_flag(scheduler_bind_type, processor_spread)*.

`s`

Same as *erlang:system_flag(scheduler_bind_type, spread)*.

`nnts`

Same as *erlang:system_flag(scheduler_bind_type, no_node_thread_spread)*.

`nnps`

Same as *erlang:system_flag(scheduler_bind_type, no_node_processor_spread)*.

`tnnps`

Same as *erlang:system_flag(scheduler_bind_type, thread_no_node_processor_spread)*.

`db`

Same as *erlang:system_flag(scheduler_bind_type, default_bind)*.

Binding of schedulers are currently only supported on newer Linux and Solaris systems.

If no CPU topology is available when the `+sbt` flag is processed and `BindType` is any other type than `u`, the runtime system will fail to start. CPU topology can be defined using the *+sct* flag. Note that the `+sct` flag may have to be passed before the `+sbt` flag on the command line (in case no CPU topology has been automatically detected).

For more information, see *erlang:system_flag(scheduler_bind_type, SchedulerBindType)*.

`+sct CpuTopology`

- `<Id> = integer(); when 0 =< <Id> =< 65535`
- `<IdRange> = <Id>-<Id>`
- `<IdOrIdRange> = <Id> | <IdRange>`
- `<IdList> = <IdOrIdRange>,<IdOrIdRange> | <IdOrIdRange>`
- `<LogicalIds> = L<IdList>`
- `<ThreadIds> = T<IdList> | t<IdList>`
- `<CoreIds> = C<IdList> | c<IdList>`
- `<ProcessorIds> = P<IdList> | p<IdList>`
- `<NodeIds> = N<IdList> | n<IdList>`
- `<IdDefs> = <LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds> | <LogicalIds><ThreadIds><CoreIds><NodeIds><ProcessorIds>`
- `CpuTopology = <IdDefs>:<IdDefs> | <IdDefs>`

Upper-case letters signify real identifiers and lower-case letters signify fake identifiers only used for description of the topology. Identifiers passed as real identifiers may be used by the runtime system when trying to access specific hardware and if they are not correct the behavior is undefined. Faked logical CPU identifiers are not accepted since there is no point in defining the CPU topology without real logical CPU identifiers. Thread, core, processor, and node identifiers may be left out. If left out, thread id defaults to `t0`, core id defaults to `c0`, processor id defaults to `p0`, and node id will be left undefined. Either each logical processor must belong to one and only one NUMA node, or no logical processors must belong to any NUMA nodes.

Both increasing and decreasing `<IdRange>`s are allowed.

NUMA node identifiers are system wide. That is, each NUMA node on the system have to have a unique identifier. Processor identifiers are also system wide. Core identifiers are processor wide. Thread identifiers are core wide.

The order of the identifier types imply the hierarchy of the CPU topology. Valid orders are either `<LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds>`, or `<LogicalIds><ThreadIds><CoreIds><NodeIds><ProcessorIds>`. That is, thread is part of a core which is part of a processor which is part of a NUMA node, or thread is part of a core which is part of a NUMA node which is part of a processor. A cpu topology can consist of both processor external, and processor internal NUMA nodes as long as each logical processor belongs to one and only one NUMA node. If `<ProcessorIds>` is left out, its default position will be before `<NodeIds>`. That is, the default is processor external NUMA nodes.

If a list of identifiers is used in an `<IdDefs>`:

- `<LogicalIds>` have to be a list of identifiers.
- At least one other identifier type apart from `<LogicalIds>` also have to have a list of identifiers.
- All lists of identifiers have to produce the same amount of identifiers.

A simple example. A single quad core processor may be described this way:

```
% erl +sct L0-3c0-3
1> erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
             {core,{logical,1}},
             {core,{logical,2}},
             {core,{logical,3}}]}]
```

A little more complicated example. Two quad core processors. Each processor in its own NUMA node. The ordering of logical processors is a little weird. This in order to give a better example of identifier lists:

```
% erl +sct L0-1,3-2c0-3p0N0:L7,4,6-5c0-3p1N1
1> erlang:system_info(cpu_topology).
[{node,[{processor,[{core,{logical,0}},
                    {core,{logical,1}},
                    {core,{logical,3}},
                    {core,{logical,2}}]}]},
 {node,[{processor,[{core,{logical,7}},
                    {core,{logical,4}},
                    {core,{logical,6}},
                    {core,{logical,5}}]}]}]
```

As long as real identifiers are correct it is okay to pass a CPU topology that is not a correct description of the CPU topology. When used with care this can actually be very useful. This in order to trick the emulator to bind its schedulers as you want. For example, if you want to run multiple Erlang runtime systems on the same machine, you want to reduce the amount of schedulers used and manipulate the CPU topology so that they bind to different logical CPUs. An example, with two Erlang runtime systems on a quad core machine:

```
% erl +sct L0-3c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname one
% erl +sct L3-0c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname two
```

In this example each runtime system have two schedulers each online, and all schedulers online will run on different cores. If we change to one scheduler online on one runtime system, and three schedulers online on the other, all schedulers online will still run on different cores.

Note that a faked CPU topology that does not reflect how the real CPU topology looks like is likely to decrease the performance of the runtime system.

For more information, see *erlang:system_flag(cpu_topology, CpuTopology)*.

`+sss size`

Suggested stack size, in kilowords, for scheduler threads. Valid range is 4-8192 kilowords. The default stack size is OS dependent.

`+t size`

Set the maximum number of atoms the VM can handle. Default is 1048576.

`+T Level`

Enables modified timing and sets the modified timing level. Currently valid range is 0-9. The timing of the runtime system will change. A high level usually means a greater change than a low level. Changing the timing can be very useful for finding timing related bugs.

Currently, modified timing affects the following:

Process spawning

A process calling `spawn`, `spawn_link`, `spawn_monitor`, or `spawn_opt` will be scheduled out immediately after completing the call. When higher modified timing levels are used, the caller will also sleep for a while after being scheduled out.

Context reductions
The amount of reductions a process is a allowed to use before being scheduled out is increased or reduced.
Input reductions
The amount of reductions performed before checking I/O is increased or reduced.

NOTE: Performance will suffer when modified timing is enabled. This flag is *only* intended for testing and debugging. Also note that `return_to` and `return_from` trace messages will be lost when tracing on the spawn BIFs. This flag may be removed or changed at any time without prior notice.

`+V`

Makes the emulator print out its version number.

`+v`

Verbose.

`+W w | i`

Sets the mapping of warning messages for `error_logger`. Messages sent to the error logger using one of the warning routines can be mapped either to errors (default), warnings (`+W w`), or info reports (`+W i`). The current mapping can be retrieved using `error_logger:warning_map/0`. See *error_logger(3)* for further information.

Environment variables

`ERL_CRASH_DUMP`

If the emulator needs to write a crash dump, the value of this variable will be the file name of the crash dump file. If the variable is not set, the name of the crash dump file will be `erl_crash.dump` in the current directory.

`ERL_CRASH_DUMP_NICE`

Unix systems: If the emulator needs to write a crash dump, it will use the value of this variable to set the nice value for the process, thus lowering its priority. The allowable range is 1 through 39 (higher values will be replaced with 39). The highest value, 39, will give the process the lowest priority.

`ERL_CRASH_DUMP_SECONDS`

Unix systems: This variable gives the number of seconds that the emulator will be allowed to spend writing a crash dump. When the given number of seconds have elapsed, the emulator will be terminated by a SIGALRM signal.

`ERL_AFLAGS`

The content of this environment variable will be added to the beginning of the command line for `erl`.

The `-extra` flag is treated specially. Its scope ends at the end of the environment variable content. Arguments following an `-extra` flag are moved on the command line into the `-extra` section, i.e. the end of the command line following after an `-extra` flag.

`ERL_ZFLAGS`and `ERL_FLAGS`

The content of these environment variables will be added to the end of the command line for `erl`.

The `-extra` flag is treated specially. Its scope ends at the end of the environment variable content. Arguments following an `-extra` flag are moved on the command line into the `-extra` section, i.e. the end of the command line following after an `-extra` flag.

`ERL_LIBS`

This environment variable contains a list of additional library directories that the code server will search for applications and add to the code path. See *code(3)*.

`ERL_EPMD_PORT`

This environment variable can contain the port number to use when communicating with *epmd*. The default port will work fine in most cases. A different port can be specified to allow nodes of independent clusters to co-exist on the same host. All nodes in a cluster must use the same epmd port number.

SEE ALSO

init(3), erl_prim_loader(3), erl_boot_server(3), code(3), application(3), heart(3), net_kernel(3), auth(3), make(3), epmd(1), erts_alloc(3)

erlc

Command

The `erlc` program provides a common way to run all compilers in the Erlang system. Depending on the extension of each input file, `erlc` will invoke the appropriate compiler. Regardless of which compiler is used, the same flags are used to provide parameters such as include paths and output directory.

The current working directory, `"."`, will not be included in the code path when running the compiler (to avoid loading Beam files from the current working directory that could potentially be in conflict with the compiler or Erlang/OTP system used by the compiler).

Exports

`erlc flags file1.ext file2.ext...`

`Erlc` compiles one or more files. The files must include the extension, for example `.erl` for Erlang source code, or `.yrl` for Yecc source code. `Erlc` uses the extension to invoke the correct compiler.

Generally Useful Flags

The following flags are supported:

-I *directory*

Instructs the compiler to search for include files in the specified directory. When encountering an `-include` or `-include_dir` directive, the compiler searches for header files in the following directories:

- `"."`, the current working directory of the file server;
- the base name of the compiled file;
- the directories specified using the `-I` option. The directory specified last is searched first.

-o *directory*

The directory where the compiler should place the output files. If not specified, output files will be placed in the current working directory.

-D*name*

Defines a macro.

-D*name=value*

Defines a macro with the given value. The value can be any Erlang term. Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms which contain tuples and list must be quoted. Terms which contain spaces must be quoted on all platforms.

-W*error*

Makes all warnings into errors.

-W*number*

Sets warning level to *number*. Default is `1`. Use `-W0` to turn off warnings.

-W

Same as `-W1`. Default.

-v

Enables verbose output.

-b *output-type*

Specifies the type of output file. Generally, *output-type* is the same as the file extension of the output file but without the period. This option will be ignored by compilers that have a a single output format.

-hybrid

Compile using the hybrid-heap emulator. This is mainly useful for compiling native code, which needs to be compiled with the same run-time system that it should be run on.

-smp

Compile using the SMP emulator. This is mainly useful for compiling native code, which needs to be compiled with the same run-time system that it should be run on.

--

Signals that no more options will follow. The rest of the arguments will be treated as file names, even if they start with hyphens.

+*term*

A flag starting with a plus ('+') rather than a hyphen will be converted to an Erlang term and passed unchanged to the compiler. For instance, the `export_all` option for the Erlang compiler can be specified as follows:

```
erlc +export_all file.erl
```

Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms which contain tuples and list must be quoted. Terms which contain spaces must be quoted on all platforms.

Special Flags

The flags in this section are useful in special situations such as re-building the OTP system.

-pa *directory*

Appends *directory* to the front of the code path in the invoked Erlang emulator. This can be used to invoke another compiler than the default one.

-pz *directory*

Appends *directory* to the code path in the invoked Erlang emulator.

Supported Compilers

.erl

Erlang source code. It generates a `.beam` file.

The options -P, -E, and -S are equivalent to +'P', +'E', and +'S', except that it is not necessary to include the single quotes to protect them from the shell.

Supported options: -I, -o, -D, -v, -W, -b.

.yrl

Yecc source code. It generates an `.erl` file.

Use the -I option with the name of a file to use that file as a customized prologue file (the `includefile` option).

Supported options: -o, -v, -I, -W (see above).

.mib

MIB for SNMP. It generates a `.bin` file.

Supported options: -I, -o, -W.

.bin

A compiled MIB for SNMP. It generates a `.hrl` file.

Supported options: -o, -v.

.rel

Script file. It generates a boot file.

Use the -I to name directories to be searched for application files (equivalent to the `path` in the option list for `systools:make_script/2`).

Supported options: -o.

.asn1

ASN1 file.

Creates an `.erl`, `.hrl`, and `.asn1db` file from an `.asn1` file. Also compiles the `.erl` using the Erlang compiler unless the `+noobj` options is given.

Supported options: -I, -o, -b, -W.

.idl

IC file.

Runs the IDL compiler.

Supported options: -I, -o.

Environment Variables

ERLC_EMULATOR

The command for starting the emulator. Default is *erl* in the same directory as the *erlc* program itself, or if it doesn't exist, *erl* in any of the directories given in the *PATH* environment variable.

SEE ALSO

erl(1), compile(3), yecc(3), snmp(3)

werl

Command

On Windows, the preferred way to start the Erlang system for interactive use is:

```
werl <arguments>
```

This will start Erlang in its own window, with fully functioning command-line editing and scrollbars. All flags except `-oldshell` work as they do for the *erl* command.

Ctrl-C is reserved for copying text to the clipboard (Ctrl-V to paste). To interrupt the runtime system or the shell process (depending on what has been specified with the +B system flag), you should use Ctrl-Break.

In cases where you want to redirect standard input and/or standard output or use Erlang in a pipeline, the `werl` is not suitable, and the `erl` program should be used instead.

The `werl` window is in many ways modelled after the `xterm` window present on other platforms, as the `xterm` model fits well with line oriented command based interaction. This means that selecting text is line oriented rather than rectangle oriented.

To select text in the `werl` window , simply press and hold the left mouse button and drag the mouse over the text you want to select. If the selection crosses line boundaries, the selected text will consist of complete lines where applicable (just like in a word processor). To select more text than fits in the window, start by selecting a small portion in the beginning of the text you want, then use the scrollbar to view the end of the desired selection, point to it and press the *right* mouse-button. The whole area between your first selection and the point where you right-clicked will be included in the selection.

The selected text is copied to the clipboard by either pressing `Ctrl-C`, using the menu or pressing the copy button in the toolbar.

Pasted text is always inserted at the current prompt position and will be interpreted by Erlang as usual keyboard input.

Previous command lines can be retrieved by pressing the `Up arrow` or by pressing `Ctrl-P`. There is also a drop down box in the toolbar containing the command history. Selecting a command in the drop down box will insert it at the prompt, just as if you used the keyboard to retrieve the command.

Closing the `werl` window will stop the Erlang emulator.

escript

Command

`escript` provides support for running short Erlang programs without having to compile them first and an easy way to retrieve the command line arguments.

Exports

`script-name script-arg1 script-arg2...`
`escript escript-flags script-name script-arg1 script-arg2...`

`escript` runs a script written in Erlang.

Here follows an example.

```
$ cat factorial
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -smp enable -sname factorial -mnesia debug verbose
main([String]) ->
    try
        N = list_to_integer(String),
        F = fac(N),
        io:format("factorial ~w = ~w\n", [N,F])
    catch
        _:_ ->
            usage()
    end;
main(_) ->
    usage().

usage() ->
    io:format("usage: factorial integer\n"),
    halt(1).

fac(0) -> 1;
fac(N) -> N * fac(N-1).
$ factorial 5
factorial 5 = 120
$ factorial
usage: factorial integer
$ factorial five
usage: factorial integer
```

The header of the Erlang script in the example differs from a normal Erlang module. The first line is intended to be the interpreter line, which invokes `escript`. However if you invoke the `escript` like this

```
$ escript factorial 5
```

the contents of the first line does not matter, but it cannot contain Erlang code as it will be ignored.

The second line in the example, contains an optional directive to the `Emacs` editor which causes it to enter the major mode for editing Erlang source files. If the directive is present it must be located on the second line.

On the third line (or second line depending on the presence of the Emacs directive), it is possible to give arguments to the emulator, such as

```
%%! -smp enable -sname factorial -mnesia debug verbose
```

Such an argument line must start with `%%!` and the rest of the line will interpreted as arguments to the emulator.

If you know the location of the `escript` executable, the first line can directly give the path to `escript`. For instance:

```
#!/usr/local/bin/escript
```

As any other kind of scripts, Erlang scripts will not work on Unix platforms if the execution bit for the script file is not set. (Use `chmod +x script-name` to turn on the execution bit.)

The rest of the Erlang script file may either contain Erlang `source code`, an `inlined beam file` or an `inlined archive file`.

An Erlang script file must always contain the function *main/1*. When the script is run, the `main/1` function will be called with a list of strings representing the arguments given to the script (not changed or interpreted in any way).

If the `main/1` function in the script returns successfully, the exit status for the script will be 0. If an exception is generated during execution, a short message will be printed and the script terminated with exit status 127.

To return your own non-zero exit code, call `halt(ExitCode)`; for instance:

```
halt(1).
```

Call `escript:script_name/0` from your to script to retrieve the pathname of the script (the pathname is usually, but not always, absolute).

If the file contains source code (as in the example above), it will be processed by the preprocessor `epp`. This means that you for example may use pre-defined macros (such as `?MODULE`) as well as include directives like the `-include_lib` directive. For instance, use

```
-include_lib("kernel/include/file.hrl").
```

to include the record definitions for the records used by the `file:read_link_info/1` function.

The script will be checked for syntactic and semantic correctness before being run. If there are warnings (such as unused variables), they will be printed and the script will still be run. If there are errors, they will be printed and the script will not be run and its exit status will be 127.

Both the module declaration and the export declaration of the `main/1` function are optional.

By default, the script will be interpreted. You can force it to be compiled by including the following line somewhere in the script file:

```
-mode(compile).
```

Execution of interpreted code is slower than compiled code. If much of the execution takes place in interpreted code it may be worthwhile to compile it, even though the compilation itself will take a little while.

As mentioned earlier, it is possible to have a script which contains precompiled `beam` code. In a precompiled script, the interpretation of the script header is exactly the same as in a script containing source code. That means that you can make a `beam` file executable by prepending the file with the lines starting with `#!` and `%%!` mentioned above. In a precompiled script, the function `main/1` must be exported.

As yet another option it is possible to have an entire Erlang archive in the script. In a archive script, the interpretation of the script header is exactly the same as in a script containing source code. That means that you can make an archive file executable by prepending the file with the lines starting with `#!` and `%%!` mentioned above. In an archive script, the function `main/1` must be exported. By default the `main/1` function in the module with the same name as the basename of the `escript` file will be invoked. This behavior can be overridden by setting the flag `-escript main Module` as one of the emulator flags. The `Module` must be the name of a module which has an exported `main/1` function. See *code(3)* for more information about archives and code loading.

In many cases it is very convenient to have a header in the escript, especially on Unix platforms. But the header is in fact optional. This means that you directly can "execute" an Erlang module, beam file or archive file without adding any header to them. But then you have to invoke the script like this:

```
$ escript factorial.erl 5
factorial 5 = 120
$ escript factorial.beam 5
factorial 5 = 120
$ escript factorial.zip 5
factorial 5 = 120
```

Options accepted by escript

-c
: Compile the escript regardless of the value of the mode attribute.

-d
: Debug the escript. Starts the debugger, loads the module containing the `main/1` function into the debugger, sets a breakpoint in `main/1` and invokes `main/1`. If the module is precompiled, it must be explicitly compiled with the `debug_info` option.

-i
: Interpret the escript regardless of the value of the mode attribute.

-s
: Only perform a syntactic and semantic check of the script file. Warnings and errors (if any) are written to the standard output, but the script will not be run. The exit status will be 0 if there were no errors, and 127 otherwise.

erlsrv

Command

This utility is specific to Windows NT/2000/XP® (and subsequent versions of Windows) It allows Erlang emulators to run as services on the Windows system, allowing embedded systems to start without any user needing to log in. The emulator started in this way can be manipulated through the Windows® services applet in a manner similar to other services.

Note that erlsrv is not a general service utility for Windows, but designed for embedded Erlang systems.

As well as being the actual service, erlsrv also provides a command line interface for registering, changing, starting and stopping services.

To manipulate services, the logged in user should have Administrator privileges on the machine. The Erlang machine itself is (default) run as the local administrator. This can be changed with the Services applet in Windows ®.

The processes created by the service can, as opposed to normal services, be "killed" with the task manager. Killing a emulator that is started by a service will trigger the "OnFail" action specified for that service, which may be a reboot.

The following parameters may be specified for each Erlang service:

- `StopAction`: This tells `erlsrv` how to stop the Erlang emulator. Default is to kill it (Win32 TerminateProcess), but this action can specify any Erlang shell command that will be executed in the emulator to make it stop. The emulator is expected to stop within 30 seconds after the command is issued in the shell. If the emulator is not stopped, it will report a running state to the service manager.
- `OnFail`: This can be either of `reboot`, `restart`, `restart_always` or `ignore` (the default). In case of `reboot`, the NT system is rebooted whenever the emulator stops (a more simple form of watchdog), this could be useful for less critical systems, otherwise use the heart functionality to accomplish this. The restart value makes the Erlang emulator be restarted (with whatever parameters are registered for the service at the occasion) when it stops. If the emulator stops again within 10 seconds, it is not restarted to avoid an infinite loop which could completely hang the NT system. `restart_always` is similar to restart, but does not try to detect cyclic restarts, it is expected that some other mechanism is present to avoid the problem. The default (ignore) just reports the service as stopped to the service manager whenever it fails, it has to be manually restarted.

  On a system where release handling is used, this should always be set to `ignore`. Use `heart` to restart the service on failure instead.
- `Machine`: The location of the Erlang emulator. The default is the `erl.exe` located in the same directory as erlsrv.exe. Do not specify `werl.exe` as this emulator, it will not work.

  If the system uses release handling, this should be set to a program similar to `start_erl.exe`.
- `Env`: Specifies an *additional* environment for the emulator. The environment variables specified here are added to the system wide environment block that is normally present when a service starts up. Variables present in both the system wide environment and in the service environment specification will be set to the value specified in the service.
- `WorkDir`: The working directory for the Erlang emulator, has to be on a local drive (there are no network drives mounted when a service starts). Default working directory for services is `%SystemDrive%%SystemPath%`. Debug log files will be placed in this directory.
- `Priority`: The process priority of the emulator, this can be one of `realtime`, `high`, `low` or `default` (the default). Real-time priority is not recommended, the machine will possibly be inaccessible to interactive users. High priority could be used if two Erlang nodes should reside on one dedicated system and one should have precedence over the other. Low process priority may be used if interactive performance should not be affected by the emulator process.

- `SName or Name`: Specifies the short or long node-name of the Erlang emulator. The Erlang services are always distributed, default is to use the service name as (short) node-name.
- `DebugType`: Can be one of `none` (default), `new`, `reuse` or `console`. Specifies that output from the Erlang shell should be sent to a "debug log". The log file is named <servicename>`.debug` or <servicename>`.debug.`<N>, where <N> is an integer between 1 and 99. The log-file is placed in the working directory of the service (as specified in WorkDir). The `reuse` option always reuses the same log file (<servicename>`.debug`) and the `new` option uses a separate log file for every invocation of the service (<servicename>`.debug.`<N>). The `console` option opens an interactive Windows® console window for the Erlang shell of the service. The `console` option automatically disables the `StopAction` and a service started with an interactive console window will not survive logouts, `OnFail` actions do not work with debug-consoles either. If no `DebugType` is specified (`none`), the output of the Erlang shell is discarded.

  The `consoleDebugType` is *not in any way* intended for production. It is *only* a convenient way to debug Erlang services during development. The `new` and `reuse` options might seem convenient to have in a production system, but one has to take into account that the logs will grow indefinitely during the systems lifetime and there is no way, short of restarting the service, to truncate those logs. In short, the `DebugType` is intended for debugging only. Logs during production are better produced with the standard Erlang logging facilities.
- `Args`: Additional arguments passed to the emulator startup program `erl.exe` (or `start_erl.exe`). Arguments that cannot be specified here are `-noinput` (StopActions would not work), `-name` and `-sname` (they are specified in any way. The most common use is for specifying cookies and flags to be passed to init:boot() (`-s`).
- `InternalServiceName`: Specifies the Windows® internal service name (not the display name, which is the one `erlsrv` uses to identify the service).

  This internal name can not be changed, it is fixed even if the service is renamed. `Erlsrv` generates a unique internal name when a service is created, it is recommended to keep to the defaut if release-handling is to be used for the application.

  The internal service name can be seen in the Windows® service manager if viewing `Properties` for an erlang service.
- `Comment`: A textual comment describing the service. Not mandatory, but shows up as the service description in the Windows® service manager.

The naming of the service in a system that uses release handling has to follow the convention *NodeName_Release*, where *NodeName* is the first part of the Erlang nodename (up to, but not including the "@") and *Release* is the current release of the application.

Exports

`erlsrv {set | add} <service-name> [<service options>]`

The set and add commands adds or modifies a Erlang service respectively. The simplest form of an add command would be completely without options in which case all default values (described above) apply. The service name is mandatory.

Every option can be given without parameters, in which case the default value is applied. Values to the options are supplied *only* when the default should not be used (i.e. `erlsrv set myservice -prio -arg` sets the default priority and removes all arguments).

The following service options are currently available:

-st[opaction] [<erlang shell command>]
: Defines the StopAction, the command given to the Erlang shell when the service is stopped. Default is none.

-on[fail] [{reboot | restart | restart_always}]
: Specifies the action to take when the Erlang emulator stops unexpectedly. Default is to ignore.

-m[achine] [<erl-command>]
: The complete path to the Erlang emulator, never use the werl program for this. Default is the `erl.exe` in the same directory as `erlsrv.exe`. When release handling is used, this should be set to a program similar to `start_erl.exe`.

-e[nv] [<variable>[=<value>]] ...
: Edits the environment block for the service. Every environment variable specified will add to the system environment block. If a variable specified here has the same name as a system wide environment variable, the specified value overrides the system wide. Environment variables are added to this list by specifying <variable>=<value> and deleted from the list by specifying <variable> alone. The environment block is automatically sorted. Any number of `-env` options can be specified in one command. Default is to use the system environment block unmodified (except for two additions, see *below*).

-w[orkdir] [<directory>]
: The initial working directory of the Erlang emulator. Default is the system directory.

-p[riority] [{low|high|realtime}]
: The priority of the Erlang emulator. The default is the Windows® default priority.

{-sn[ame] | -n[ame]} [<node-name>]
: The node-name of the Erlang machine, distribution is mandatory. Default is `-sname <service name>`.

-d[ebugtype] [{new|reuse|console}]
: Specifies where shell output should be sent, default is that shell output is discarded. To be used only for debugging.

-ar[gs] [<limited erl arguments>]
: Additional arguments to the Erlang emulator, avoid `-noinput`, `-noshell` and `-sname`/`-name`. Default is no additional arguments. Remember that the services cookie file is not necessarily the same as the interactive users. The service runs as the local administrator. All arguments should be given together in one string, use double quotes (") to give an argument string containing spaces and use quoted quotes (\") to give an quote within the argument string if necessary.

-i[nternalservicename] [<internal name>]
: *Only* allowed for `add`. Specifies a Windows® internal service name for the service, which by default is set to something unique (prefixed with the original service name) by erlsrv when adding a new service. Specifying this is a purely cosmethic action and is *not* recommended if release handling is to be performed. The internal service name cannot be changed once the service is created. The internal name is *not* to be confused with the ordinary service name, which is the name used to identify a service to erlsrv.

-c[omment] [<short description>]
: Specifies a textual comment describing the service. This comment will show upp as the service description in the Windows® service manager.

`erlsrv {start | stop | disable | enable} <service-name>`

These commands are only added for convenience, the normal way to manipulate the state of a service is through the control panels services applet. The `start` and `stop` commands communicates with the service manager for stopping and starting a service. The commands wait until the service is actually stopped or started. When disabling a service, it is not stopped, the disabled state will not take effect until the service actually is stopped. Enabling a service sets it in automatic mode, that is started at boot. This command cannot set the service to manual.

`erlsrv remove <service-name>`

This command removes the service completely with all its registered options. It will be stopped before it is removed.

`erlsrv list [<service-name>]`

If no service name is supplied, a brief listing of all Erlang services is presented. If a service-name is supplied, all options for that service are presented.

erlsrv help

ENVIRONMENT

The environment of an Erlang machine started as a service will contain two special variables, ERLSRV_SERVICE_NAME, which is the name of the service that started the machine and ERLSRV_EXECUTABLE which is the full path to the erlsrv.exe that can be used to manipulate the service. This will come in handy when defining a heart command for your service. A command file for restarting a service will simply look like this:

```
@echo off
%ERLSRV_EXECUTABLE% stop %ERLSRV_SERVICE_NAME%
%ERLSRV_EXECUTABLE% start %ERLSRV_SERVICE_NAME%
```

This command file is then set as heart command.

The environment variables can also be used to detect that we are running as a service and make port programs react correctly to the control events generated on logout (see below).

PORT PROGRAMS

When a program runs in the service context, it has to handle the control events that is sent to every program in the system when the interactive user logs off. This is done in different ways for programs running in the console subsystem and programs running as window applications. An application which runs in the console subsystem (normal for port programs) uses the win32 function SetConsoleCtrlHandler to a control handler that returns TRUE in answer to the CTRL_LOGOFF_EVENT. Other applications just forward WM_ENDSESSION and WM_QUERYENDSESSION to the default window procedure. Here is a brief example in C of how to set the console control handler:

```
#include <windows.h>
/*
** A Console control handler that ignores the log off events,
** and lets the default handler take care of other events.
*/
BOOL WINAPI service_aware_handler(DWORD ctrl){
    if(ctrl == CTRL_LOGOFF_EVENT)
        return TRUE;
    return FALSE;
}

void initialize_handler(void){
    char buffer[2];
    /*
     * We assume we are running as a service if this
     * environment variable is defined
     */
    if(GetEnvironmentVariable("ERLSRV_SERVICE_NAME",buffer,
                              (DWORD) 2)){
        /*
        ** Actually set the control handler
        */
        SetConsoleCtrlHandler(&service_aware_handler, TRUE);
    }
}
```

NOTES

Even though the options are described in a Unix-like format, the case of the options or commands is not relevant, and the "/" character for options can be used as well as the "-" character.

Note that the program resides in the emulators `bin`-directory, not in the `bin`-directory directly under the Erlang root. The reasons for this are the subtle problem of upgrading the emulator on a running system, where a new version of the runtime system should not need to overwrite existing (and probably used) executables.

To easily manipulate the Erlang services, put the `<erlang_root>\erts-<version>\bin` directory in the path instead of `<erlang_root>\bin`. The erlsrv program can be found from inside Erlang by using the `os:find_executable/1` Erlang function.

For release handling to work, use `start_erl` as the Erlang machine. It is also worth mentioning again that the name of the service is significant (see *above*).

SEE ALSO

start_erl(1), release_handler(3)

start_erl

Command

This describes the `start_erl` program specific to Windows NT. Although there exists programs with the same name on other platforms, their functionality is not the same.

The `start_erl` program is distributed both in compiled form (under <Erlang root>\\erts-<version>\\bin) and in source form (under <Erlang root>\\erts-<version>\\src). The purpose of the source code is to make it possible to easily customize the program for local needs, such as cyclic restart detection etc. There is also a "make"-file, written for the `nmake` program distributed with Microsoft® Visual C++®. The program can however be compiled with any Win32 C compiler (possibly with slight modifications).

The purpose of the program is to aid release handling on Windows NT®. The program should be called by the `erlsrv` program, read up the release data file start_erl.data and start Erlang. Certain options to start_erl are added and removed by the release handler during upgrade with emulator restart (more specifically the `-data` option).

Exports

`start_erl [<erl options>] ++ [<start_erl options>]`

The `start_erl` program in its original form recognizes the following options:

++
: Mandatory, delimits start_erl options from normal Erlang options. Everything on the command line *before* the ++ is interpreted as options to be sent to the `erl` program. Everything *after*++ is interpreted as options to `start_erl` itself.

-reldir <release root>
: Mandatory if the environment variable `RELDIR` is not specified. Tells start_erl where the root of the release tree is placed in the file-system (like <Erlang root>\\releases). The `start_erl.data` file is expected to be placed in this directory (if not otherwise specified).

-data <data file name>
: Optional, specifies another data file than start_erl.data in the <release root>. It is specified relative to the <release root> or absolute (including drive letter etc.). This option is used by the release handler during upgrade and should not be used during normal operation. The release data file should not normally be named differently.

-bootflags <boot flags file name>
: Optional, specifies a file name relative to actual release directory (that is the subdirectory of <release root> where the `.boot` file etc. are placed). The contents of this file is appended to the command line when Erlang is started. This makes it easy to start the emulator with different options for different releases.

NOTES

As the source code is distributed, it can easily be modified to accept other options. The program must still accept the `-data` option with the semantics described above for the release handler to work correctly.

The Erlang emulator is found by examining the registry keys for the emulator version specified in the release data file. The new emulator needs to be properly installed before the upgrade for this to work.

Although the program is located together with files specific to emulator version, it is not expected to be specific to the emulator version. The release handler does *not* change the `-machine` option to `erlsrv` during emulator restart. Place the (possibly customized) `start_erl` program so that it is not overwritten during upgrade.

The `erlsrv` program's default options are not sufficient for release handling. The machine `erlsrv` starts should be specified as the `start_erl` program and the arguments should contain the ++ followed by desired options.

SEE ALSO

erlsrv(1), release_handler(3)

erl_set_memory_block

C Library

This documentation is specific to VxWorks.

The `erl_set_memory_block` function/command initiates custom memory allocation for the Erlang emulator. It has to be called before the Erlang emulator is started and makes Erlang use one single large memory block for all memory allocation.

The memory within the block can be utilized by other tasks than Erlang. This is accomplished by calling the functions `sys_alloc`, `sys_realloc` and `sys_free` instead of `malloc`, `realloc` and `free` respectively.

The purpose of this is to avoid problems inherent in the VxWorks systems `malloc` library. The memory allocation within the large memory block avoids fragmentation by using an "address order first fit" algorithm. Another advantage of using a separate memory block is that resource reclamation can be made more easily when Erlang is stopped.

The `erl_set_memory_block` function is callable from any C program as an ordinary 10 argument function as well as from the commandline.

Exports

`int erl_set_memory_block(size_t size, void *ptr, int warn_mixed_malloc, int realloc_always_moves, int use_reclaim, ...)`

The function is called before Erlang is started to specify a large memory block where Erlang can maintain memory internally.

Parameters:

size_t size
: The size in bytes of Erlang's internal memory block. Has to be specified. Note that the VxWorks system uses dynamic memory allocation heavily, so leave some memory to the system.

void *ptr
: A pointer to the actual memory block of size `size`. If this is specified as 0 (NULL), Erlang will allocate the memory when starting and will reclaim the memory block (as a whole) when stopped.

  If a memory block is allocated and provided here, the `sys_alloc` etc routines can still be used after the Erlang emulator is stopped. The Erlang emulator can also be restarted while other tasks using the memory block are running without destroying the memory. If Erlang is to be restarted, also set the `use_reclaim` flag.

  If 0 is specified here, the Erlang system should not be stopped while some other task uses the memory block (has called `sys_alloc`).

int warn_mixed_malloc
: If this flag is set to true (anything else than 0), the system will write a warning message on the console if a program is mixing normal `malloc` with `sys_realloc` or `sys_free`.

int realloc_always_moves
: If this flag is set to true (anything else than 0), all calls to `sys_realloc` result in a moved memory block. This can in certain conditions give less fragmentation. This flag may be removed in future releases.

int use_reclaim
: If this flag is set to true (anything else than 0), all memory allocated with `sys_alloc` is automatically reclaimed as soon as a task exits. This is very useful to make writing port programs (and other programs as well) easier.

Combine this with using the routines `save_open` etc. specified in the reclaim.h file delivered in the Erlang distribution.

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0.

`int erl_memory_show(...)`

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0.

`int erl_mem_info_get(MEM_PART_STATS *stats)`

Parameter:

MEM_PART_STATS *stats
A pointer to a MEM_PART_STATS structure as defined in `<memLib.h>`. A successful call will fill in all fields of the structure, on error all fields are left untouched.

Return Value:

Returns 0 (OK) on success, otherwise a value <> 0

NOTES

The memory block used by Erlang actually does not need to be inside the area known to ordinary `malloc`. It is possible to set the `USER_RESERVED_MEM` preprocessor symbol when compiling the wind kernel and then use user reserved memory for Erlang. Erlang can therefor utilize memory above the 32 Mb limit of VxWorks on the PowerPC architecture.

Example:

In config.h for the wind kernel:

```
#undef LOCAL_MEM_AUTOSIZE
#undef LOCAL_MEM_SIZE
#undef USER_RESERVED_MEM

#define LOCAL_MEM_SIZE        0x05000000
#define USER_RESERVED_MEM     0x03000000
```

In the start-up script/code for the VxWorks node:

```
erl_set_memory_block(sysPhysMemTop()-sysMemTop(),sysMemTop(),0,0,1);
```

Setting the `use_reclaim` flag decreases performance of the system, but makes programming much easier. Other similar facilities are present in the Erlang system even without using a separate memory block. The routines called `save_malloc`, `save_realloc` and `save_free` provide the same facilities by using VxWorks own `malloc`. Similar routines exist for files, see the file `reclaim.h` in the distribution.

run_erl

Command

This describes the `run_erl` program specific to Solaris/Linux. This program redirect the standard input and standard output streams so that all output can be logged. It also let the program `to_erl` connect to the Erlang console making it possible to monitor and debug an embedded system remotely.

You can read more about the use in the `Embedded System User's Guide`.

Exports

`run_erl [-daemon] pipe_dir/ log_dir "exec command [command_arguments]"`

The `run_erl` program arguments are:

-daemon
: This option is highly recommended. It makes run_erl run in the background completely detached from any controlling terminal and the command returns to the caller immediately. Without this option, run_erl must be started using several tricks in the shell to detach it completely from the terminal in use when starting it. The option must be the first argument to run_erl on the command line.

pipe_dir
: This is where to put the named pipe, usually `/tmp/`. It shall be suffixed by a `/` (slash), i.e. not `/tmp/epipies`, but `/tmp/epipes/`.

log_dir
: This is where the log files are written. There will be one log file, `run_erl.log` that log progress and warnings from the `run_erl` program itself and there will be up to five log files at maximum 100KB each (both number of logs and sizes can be changed by environment variables, see below) with the content of the standard streams from and to the command. When the logs are full `run_erl` will delete and reuse the oldest log file.

"exec command [command_arguments]"
: In the third argument `command` is the to execute where everything written to stdin and stdout is logged to `log_dir`.

Notes concerning the log files

While running, run_erl (as stated earlier) sends all output, uninterpreted, to a log file. The file is called `erlang.log.N`, where N is a number. When the log is "full", default after 100KB, run_erl starts to log in file `erlang.log.(N+1)`, until N reaches a certain number (default 5), where after N starts at 1 again and the oldest files start getting overwritten. If no output comes from the erlang shell, but the erlang machine still seems to be alive, an "ALIVE" message is written to the log, it is a timestamp and is written, by default, after 15 minutes of inactivity. Also, if output from erlang is logged but it's been more than 5 minutes (default) since last time we got anything from erlang, a timestamp is written in the log. The "ALIVE" messages look like this:

```
===== ALIVE <date-time-string>
```

while the other timestamps look like this:

```
===== <date-time-string>
```

The `date-time-string` is the date and time the message is written, default in local time (can be changed to GMT if one wants to) and is formatted with the ANSI-C function `strftime` using the format string `%a %b %e %T %Z %Y`, which produces messages on the line of `===== ALIVE Thu May 15 10:13:36 MEST 2003`, this can be changed, see below.

Environment variables

The following environment variables are recognized by run_erl and change the logging behavior. Also see the notes above to get more info on how the log behaves.

RUN_ERL_LOG_ALIVE_MINUTES
: How long to wait for output (in minutes) before writing an "ALIVE" message to the log. Default is 15, can never be less than 1.

RUN_ERL_LOG_ACTIVITY_MINUTES
: How long erlang need to be inactive before output will be preceded with a timestamp. Default is RUN_ERL_LOG_ALIVE_MINUTES div 3, but never less than 1.

RUN_ERL_LOG_ALIVE_FORMAT
: Specifies another format string to be used in the strftime C library call. i.e specifying this to `"%e-%b-%Y, %T %Z"` will give log messages with timestamps looking like `15-May-2003, 10:23:04 MET` etc. See the documentation for the C library function strftime for more information. Default is `"%a %b %e %T %Z %Y"`.

RUN_ERL_LOG_ALIVE_IN_UTC
: If set to anything else than "0", it will make all times displayed by run_erl to be in UTC (GMT,CET,MET, without DST), rather than in local time. This does not affect data coming from erlang, only the logs output directly by run_erl. The application `sasl` can be modified accordingly by setting the erlang application variable `utc_log` to `true`.

RUN_ERL_LOG_GENERATIONS
: Controls the number of log files written before older files are being reused. Default is 5, minimum is 2, maximum is 1000.

RUN_ERL_LOG_MAXSIZE
: The size (in bytes) of a log file before switching to a new log file. Default is 100000, minimum is 1000 and maximum is approximately 2^30.

SEE ALSO

start(1), start_erl(1)

start

Command

This describes the `start` script that is an example script on how to startup the Erlang system in embedded mode on Unix.

You can read more about the use in the `Embedded System User's Guide`.

Exports

`start [ data_file ]`

In the example there is one argument

data_file
: Optional, specifies what `start_erl.data` file to use.

There is also an environment variable `RELDIR` that can be set prior to calling this example that set the directory where to find the release files.

SEE ALSO

run_erl(1), start_erl(1)

erl_driver

C Library

As of erts version 5.5.3 the driver interface has been extended (see *extended marker*). The extended interface introduce *version management*, the possibility to pass capability flags (see *driver flags*) to the runtime system at driver initialization, and some new driver API functions.

> ### Note:
>
> Old drivers (compiled with an `erl_driver.h` from an earlier erts version than 5.5.3) have to be recompiled (but does not have to use the extended interface).

The driver calls back to the emulator, using the API functions declared in `erl_driver.h`. They are used for outputting data from the driver, using timers, etc.

A driver is a library with a set of function that the emulator calls, in response to Erlang functions and message sending. There may be multiple instances of a driver, each instance is connected to an Erlang port. Every port has a port owner process. Communication with the port is normally done through the port owner process.

Most of the functions takes the `port` handle as an argument. This identifies the driver instance. Note that this port handle must be stored by the driver, it is not given when the driver is called from the emulator (see *driver_entry*).

Some of the functions takes a parameter of type `ErlDrvBinary`, a driver binary. It should be both allocated and freed by the caller. Using a binary directly avoid one extra copying of data.

Many of the output functions has a "header buffer", with `hbuf` and `hlen` parameters. This buffer is sent as a list before the binary (or list, depending on port mode) that is sent. This is convenient when matching on messages received from the port. (Although in the latest versions of Erlang, there is the binary syntax, that enables you to match on the beginning of a binary.)

In the runtime system with SMP support, drivers are locked either on driver level or port level (driver instance level). By default driver level locking will be used, i.e., only one emulator thread will execute code in the driver at a time. If port level locking is used, multiple emulator threads may execute code in the driver at the same time. There will only be one thread at a time calling driver call-backs corresponding to the same port, though. In order to enable port level locking set the `ERL_DRV_FLAG_USE_PORT_LOCKING` *driver flag* in the *driver_entry* used by the driver. When port level locking is used it is the responsibility of the driver writer to synchronize all accesses to data shared by the ports (driver instances).

Most drivers written before the runtime system with SMP support existed will be able to run in the runtime system with SMP support without being rewritten if driver level locking is used.

> ### Note:
>
> It is assumed that drivers does not access other drivers. If drivers should access each other they have to provide their own mechanism for thread safe synchronization. Such "inter driver communication" is strongly discouraged.

Previously, in the runtime system without SMP support, specific driver call-backs were always called from the same thread. This is *not* the case in the runtime system with SMP support. Regardless of locking scheme used, calls to driver call-backs may be made from different threads, e.g., two consecutive calls to exactly the same call-back for exactly the same port may be made from two different threads. This will for *most* drivers not be a problem, but it might. Drivers

that depend on all call-backs being called in the same thread, *have* to be rewritten before being used in the runtime system with SMP support.

> Note:
>
> Regardless of locking scheme used, calls to driver call-backs may be made from different threads.

Most functions in this API are *not* thread-safe, i.e., they may *not* be called from an arbitrary thread. Function that are not documented as thread-safe may only be called from driver call-backs or function calls descending from a driver call-back call. Note that driver call-backs may be called from different threads. This, however, is not a problem for any functions in this API, since the emulator have control over these threads.

> Note:
>
> Functions not explicitly documented as thread-safe are *not* thread-safe. Also note that some functions are *only* thread safe when used in a runtime system with SMP support.

FUNCTIONALITY

All functions that a driver needs to do with Erlang are performed through driver API functions. There are functions for the following functionality:

Timer functions
: Timer functions are used to control the timer that a driver may use. The timer will have the emulator call the *timeout* entry function after a specified time. Only one timer is available for each driver instance.

Queue handling
: Every driver instance has an associated queue. This queue is a `SysIOVec` that works as a buffer. It's mostly used for the driver to buffer data that should be written to a device, it is a byte stream. If the port owner process closes the driver, and the queue is not empty, the driver will not be closed. This enables the driver to flush its buffers before closing.

  The queue can be manipulated from arbitrary threads if a port data lock is used. See documentation of the *ErlDrvPDL* type for more information.

Output functions
: With the output functions, the driver sends data back the emulator. They will be received as messages by the port owner process, see `open_port/2`. The vector function and the function taking a driver binary is faster, because that avoid copying the data buffer. There is also a fast way of sending terms from the driver, without going through the binary term format.

Failure
: The driver can exit and signal errors up to Erlang. This is only for severe errors, when the driver can't possibly keep open.

Asynchronous calls
: The latest Erlang versions (R7B and later) has provision for asynchronous function calls, using a thread pool provided by Erlang. There is also a select call, that can be used for asynchronous drivers.

Multi-threading
: A POSIX thread like API for multi-threading is provided. The Erlang driver thread API only provide a subset of the functionality provided by the POSIX thread API. The subset provided is more or less the basic functionality needed for multi-threaded programming:

- *Threads*
- *Mutexes*
- *Condition variables*
- *Read/Write locks*
- *Thread specific data*

The Erlang driver thread API can be used in conjunction with the POSIX thread API on UN-ices and with the Windows native thread API on Windows. The Erlang driver thread API has the advantage of being portable, but there might exist situations where you want to use functionality from the POSIX thread API or the Windows native thread API.

The Erlang driver thread API only return error codes when it is reasonable to recover from an error condition. If it isn't reasonable to recover from an error condition, the whole runtime system is terminated. For example, if a create mutex operation fails, an error code is returned, but if a lock operation on a mutex fails, the whole runtime system is terminated.

Note that there exist no "condition variable wait with timeout" in the Erlang driver thread API. This is due to issues with `pthread_cond_timedwait()`. When the system clock suddenly is changed, it isn't always guaranteed that you will wake up from the call as expected. An Erlang runtime system has to be able to cope with sudden changes of the system clock. Therefore, we have omitted it from the Erlang driver thread API. In the Erlang driver case, timeouts can and should be handled with the timer functionality of the Erlang driver API.

In order for the Erlang driver thread API to function, thread support has to be enabled in the runtime system. An Erlang driver can check if thread support is enabled by use of *driver_system_info()*. Note that some functions in the Erlang driver API are thread-safe only when the runtime system has SMP support, also this information can be retrieved via *driver_system_info()*. Also note that a lot of functions in the Erlang driver API are *not* thread-safe regardless of whether SMP support is enabled or not. If a function isn't documented as thread-safe it is *not* thread-safe.

NOTE: When executing in an emulator thread, it is *very important* that you unlock *all* locks you have locked before letting the thread out of your control; otherwise, you are *very likely* to deadlock the whole emulator. If you need to use thread specific data in an emulator thread, only have the thread specific data set while the thread is under your control, and clear the thread specific data before you let the thread out of your control.

In the future there will probably be debug functionality integrated with the Erlang driver thread API. All functions that create entities take a `name` argument. Currently the `name` argument is unused, but it will be used when the debug functionality has been implemented. If you name all entities created well, the debug functionality will be able to give you better error reports.

Adding / remove drivers
: A driver can add and later remove drivers.

Monitoring processes
: A driver can monitor a process that does not own a port.

Version management

Version management is enabled for drivers that have set the *extended_marker* field of their *driver_entry* to `ERL_DRV_EXTENDED_MARKER`. `erl_driver.h` defines `ERL_DRV_EXTENDED_MARKER`, `ERL_DRV_EXTENDED_MAJOR_VERSION`, and `ERL_DRV_EXTENDED_MINOR_VERSION`. `ERL_DRV_EXTENDED_MAJOR_VERSION` will be incremented when driver incompatible changes are made to the Erlang runtime system. Normally it will suffice to recompile drivers when the `ERL_DRV_EXTENDED_MAJOR_VERSION` has changed, but it could, under rare circumstances, mean that drivers have to be slightly modified. If so, this will of course be documented. `ERL_DRV_EXTENDED_MINOR_VERSION` will be incremented when new features are added. The runtime system use the minor version of the driver to determine what features to use. The runtime system will refuse to load a driver if the major versions differ, or if the major versions are equal and the minor version used by the driver is greater than the one used by the runtime system.

The emulator tries to check that a driver that doesn't use the extended driver interface isn't incompatible when loading it. It can, however, not make sure that it isn't incompatible. Therefore, when loading a driver that doesn't use the extended driver interface, there is a risk that it will be loaded also when the driver is incompatible. When the driver use the extended driver interface, the emulator can verify that it isn't of an incompatible driver version. You are therefore advised to use the extended driver interface.

DATA TYPES

ErlDrvSysInfo

```
typedef struct ErlDrvSysInfo {
   int driver_major_version;
   int driver_minor_version;
   char *erts_version;
   char *otp_release;
   int thread_support;
   int smp_support;
   int async_threads;
   int scheduler_threads;
   int nif_major_version;
   int nif_minor_version;
} ErlDrvSysInfo;
```

The `ErlDrvSysInfo` structure is used for storage of information about the Erlang runtime system. *driver_system_info()* will write the system information when passed a reference to a `ErlDrvSysInfo` structure. A description of the fields in the structure follow:

`driver_major_version`
The value of *ERL_DRV_EXTENDED_MAJOR_VERSION* when the runtime system was compiled. This value is the same as the value of *ERL_DRV_EXTENDED_MAJOR_VERSION* used when compiling the driver; otherwise, the runtime system would have refused to load the driver.

`driver_minor_version`
The value of *ERL_DRV_EXTENDED_MINOR_VERSION* when the runtime system was compiled. This value might differ from the value of *ERL_DRV_EXTENDED_MINOR_VERSION* used when compiling the driver.

`erts_version`
A string containing the version number of the runtime system (the same as returned by *erlang:system_info(version)*).

`otp_release`
A string containing the OTP release number (the same as returned by *erlang:system_info(otp_release)*).

`thread_support`
A value `!= 0` if the runtime system has thread support; otherwise, `0`.

`smp_support`
A value `!= 0` if the runtime system has SMP support; otherwise, `0`.

`thread_support`
A value `!= 0` if the runtime system has thread support; otherwise, `0`.

`smp_support`
A value `!= 0` if the runtime system has SMP support; otherwise, `0`.

`async_threads`
The number of async threads in the async thread pool used by *driver_async()* (the same as returned by *erlang:system_info(thread_pool_size)*).

`scheduler_threads`
The number of scheduler threads used by the runtime system (the same as returned by *erlang:system_info(schedulers)*).

`nif_major_version`
The value of `ERL_NIF_MAJOR_VERSION` when the runtime system was compiled.
`nif_minor_version`
The value of `ERL_NIF_MINOR_VERSION` when the runtime system was compiled.

ErlDrvBinary

```
typedef struct ErlDrvBinary {
   int orig_size;
   char orig_bytes[];
} ErlDrvBinary;
```

The `ErlDrvBinary` structure is a binary, as sent between the emulator and the driver. All binaries are reference counted; when `driver_binary_free` is called, the reference count is decremented, when it reaches zero, the binary is deallocated. The `orig_size` is the size of the binary, and `orig_bytes` is the buffer. The `ErlDrvBinary` does not have a fixed size, its size is `orig_size + 2 * sizeof(int)`.

> **Note:**
>
> The `refc` field has been removed. The reference count of an `ErlDrvBinary` is now stored elsewhere. The reference count of an `ErlDrvBinary` can be accessed via *driver_binary_get_refc()*, *driver_binary_inc_refc()*, and *driver_binary_dec_refc()*.

Some driver calls, such as `driver_enq_binary`, increments the driver reference count, and others, such as `driver_deq` decrements it.

Using a driver binary instead of a normal buffer, is often faster, since the emulator doesn't need to copy the data, only the pointer is used.

A driver binary allocated in the driver, with `driver_alloc_binary`, should be freed in the driver (unless otherwise stated), with `driver_free_binary`. (Note that this doesn't necessarily deallocate it, if the driver is still referred in the emulator, the ref-count will not go to zero.)

Driver binaries are used in the `driver_output2` and `driver_outputv` calls, and in the queue. Also the driver call-back *outputv* uses driver binaries.

If the driver of some reason or another, wants to keep a driver binary around, in a static variable for instance, the reference count should be incremented, and the binary can later be freed in the *stop* call-back, with `driver_free_binary`.

Note that since a driver binary is shared by the driver and the emulator, a binary received from the emulator or sent to the emulator, must not be changed by the driver.

From erts version 5.5 (OTP release R11B), orig_bytes is guaranteed to be properly aligned for storage of an array of doubles (usually 8-byte aligned).

ErlDrvData

The `ErlDrvData` is a handle to driver-specific data, passed to the driver call-backs. It is a pointer, and is most often casted to a specific pointer in the driver.

SysIOVec

This is a system I/O vector, as used by `writev` on unix and `WSASend` on Win32. It is used in `ErlIOVec`.

ErlIOVec

```
typedef struct ErlIOVec {
  int vsize;
  int size;
  SysIOVec* iov;
  >ErlDrvBinary** binv;
} ErlIOVec;
```

The I/O vector used by the emulator and drivers, is a list of binaries, with a `SysIOVec` pointing to the buffers of the binaries. It is used in `driver_outputv` and the *outputv* driver call-back. Also, the driver queue is an `ErlIOVec`.

ErlDrvMonitor

When a driver creates a monitor for a process, a `ErlDrvMonitor` is filled in. This is an opaque data-type which can be assigned to but not compared without using the supplied compare function (i.e. it behaves like a struct).

The driver writer should provide the memory for storing the monitor when calling *driver_monitor_process*. The address of the data is not stored outside of the driver, so the `ErlDrvMonitor` can be used as any other datum, it can be copied, moved in memory, forgotten etc.

ErlDrvNowData

The `ErlDrvNowData` structure holds a timestamp consisting of three values measured from some arbitrary point in the past. The three structure members are:

megasecs
The number of whole megaseconds elapsed since the arbitrary point in time
secs
The number of whole seconds elapsed since the arbitrary point in time
microsecs
The number of whole microseconds elapsed since the arbitrary point in time

ErlDrvPDL

If certain port specific data have to be accessed from other threads than those calling the driver call-backs, a port data lock can be used in order to synchronize the operations on the data. Currently, the only port specific data that the emulator associates with the port data lock is the driver queue.

Normally a driver instance does not have a port data lock. If the driver instance want to use a port data lock, it has to create the port data lock by calling *driver_pdl_create()*. *NOTE*: Once the port data lock has been created, every access to data associated with the port data lock have to be done while having the port data lock locked. The port data lock is locked, and unlocked, respectively, by use of *driver_pdl_lock()*, and *driver_pdl_unlock()*.

A port data lock is reference counted, and when the reference count reach zero, it will be destroyed. The emulator will at least increment the reference count once when the lock is created and decrement it once when the port associated with the lock terminates. The emulator will also increment the reference count when an async job is enqueued and decrement it after an async job has been invoked, or canceled. Besides this, it is the responsibility of the driver to ensure that the reference count does not reach zero before the last use of the lock by the driver has been made. The reference count can be read, incremented, and decremented, respectively, by use of *driver_pdl_get_refc()*, *driver_pdl_inc_refc()*, and *driver_pdl_dec_refc()*.

ErlDrvTid

Thread identifier.

See also: *erl_drv_thread_create()*, *erl_drv_thread_exit()*, *erl_drv_thread_join()*, *erl_drv_thread_self()*, and *erl_drv_equal_tids()*.

ErlDrvThreadOpts

```
int suggested_stack_size;
```

Thread options structure passed to *erl_drv_thread_create()*. Currently the following fields exist:

suggested_stack_size
A suggestion, in kilo-words, on how large stack to use. A value less than zero means default size.

See also: *erl_drv_thread_opts_create()*, *erl_drv_thread_opts_destroy()*, and *erl_drv_thread_create()*.

ErlDrvMutex

Mutual exclusion lock. Used for synchronizing access to shared data. Only one thread at a time can lock a mutex.

See also: *erl_drv_mutex_create()*, *erl_drv_mutex_destroy()*, *erl_drv_mutex_lock()*, *erl_drv_mutex_trylock()*, and *erl_drv_mutex_unlock()*.

ErlDrvCond

Condition variable. Used when threads need to wait for a specific condition to appear before continuing execution. Condition variables need to be used with associated mutexes.

See also: *erl_drv_cond_create()*, *erl_drv_cond_destroy()*, *erl_drv_cond_signal()*, *erl_drv_cond_broadcast()*, and *erl_drv_cond_wait()*.

ErlDrvRWLock

Read/write lock. Used to allow multiple threads to read shared data while only allowing one thread to write the same data. Multiple threads can read lock an rwlock at the same time, while only one thread can read/write lock an rwlock at a time.

See also: *erl_drv_rwlock_create()*, *erl_drv_rwlock_destroy()*, *erl_drv_rwlock_rlock()*, *erl_drv_rwlock_tryrlock()*, *erl_drv_rwlock_runlock()*, *erl_drv_rwlock_rwlock()*, *erl_drv_rwlock_tryrwlock()*, and *erl_drv_rwlock_rwunlock()*.

ErlDrvTSDKey

Key which thread specific data can be associated with.

See also: *erl_drv_tsd_key_create()*, *erl_drv_tsd_key_destroy()*, *erl_drv_tsd_set()*, and *erl_drv_tsd_get()*.

Exports

`void driver_system_info(ErlDrvSysInfo *sys_info_ptr, size_t size)`

This function will write information about the Erlang runtime system into the *ErlDrvSysInfo* structure referred to by the first argument. The second argument should be the size of the *ErlDrvSysInfo* structure, i.e., `sizeof(ErlDrvSysInfo)`.

See the documentation of the *ErlDrvSysInfo* structure for information about specific fields.

`int driver_output(ErlDrvPort port, char *buf, int len)`

The `driver_output` function is used to send data from the driver up to the emulator. The data will be received as terms or binary data, depending on how the driver port was opened.

The data is queued in the port owner process' message queue. Note that this does not yield to the emulator. (Since the driver and the emulator runs in the same thread.)

The parameter `buf` points to the data to send, and `len` is the number of bytes.

The return value for all output functions is 0. (Unless the driver is used for distribution, in which case it can fail and return -1. For normal use, the output function always returns 0.)

int driver_output2(ErlDrvPort port, char *hbuf, int hlen, char *buf, int len)

The `driver_output2` function first sends `hbuf` (length in `hlen`) data as a list, regardless of port settings. Then `buf` is sent as a binary or list. E.g. if `hlen` is 3 then the port owner process will receive `[H1, H2, H3 | T]`.

The point of sending data as a list header, is to facilitate matching on the data received.

The return value is 0 for normal use.

int driver_output_binary(ErlDrvPort port, char *hbuf, int hlen, ErlDrvBinary* bin, int offset, int len)

This function sends data to port owner process from a driver binary, it has a header buffer (`hbuf` and `hlen`) just like `driver_output2`. The `hbuf` parameter can be `NULL`.

The parameter `offset` is an offset into the binary and `len` is the number of bytes to send.

Driver binaries are created with `driver_alloc_binary`.

The data in the header is sent as a list and the binary as an Erlang binary in the tail of the list.

E.g. if `hlen` is 2, then the port owner process will receive `[H1, H2 | <<T>>]`.

The return value is 0 for normal use.

Note that, using the binary syntax in Erlang, the driver application can match the header directly from the binary, so the header can be put in the binary, and hlen can be set to 0.

int driver_outputv(ErlDrvPort port, char* hbuf, int hlen, ErlIOVec *ev, int skip)

This function sends data from an IO vector, `ev`, to the port owner process. It has a header buffer (`hbuf` and `hlen`), just like `driver_output2`.

The `skip` parameter is a number of bytes to skip of the `ev` vector from the head.

You get vectors of `ErlIOVec` type from the driver queue (see below), and the *outputv* driver entry function. You can also make them yourself, if you want to send several `ErlDrvBinary` buffers at once. Often it is faster to use `driver_output` or `driver_output_binary`.

E.g. if `hlen` is 2 and `ev` points to an array of three binaries, the port owner process will receive `[H1, H2, <<B1>>, <<B2>> | <<B3>>]`.

The return value is 0 for normal use.

The comment for `driver_output_binary` applies for `driver_outputv` too.

int driver_vec_to_buf(ErlIOVec *ev, char *buf, int len)

This function collects several segments of data, referenced by `ev`, by copying them in order to the buffer `buf`, of the size `len`.

If the data is to be sent from the driver to the port owner process, it is faster to use `driver_outputv`.

The return value is the space left in the buffer, i.e. if the `ev` contains less than `len` bytes it's the difference, and if `ev` contains `len` bytes or more, it's 0. This is faster if there is more than one header byte, since the binary syntax can construct integers directly from the binary.

int driver_set_timer(ErlDrvPort port, unsigned long time)

This function sets a timer on the driver, which will count down and call the driver when it is timed out. The `time` parameter is the time in milliseconds before the timer expires.

When the timer reaches 0 and expires, the driver entry function *timeout* is called.

Note that there is only one timer on each driver instance; setting a new timer will replace an older one.

Return value i 0 (-1 only when the `timeout` driver function is `NULL`).

int driver_cancel_timer(ErlDrvPort port)

This function cancels a timer set with `driver_set_timer`.

The return value is 0.

int driver_read_timer(ErlDrvPort port, unsigned long *time_left)

This function reads the current time of a timer, and places the result in `time_left`. This is the time in milliseconds, before the timeout will occur.

The return value is 0.

int driver_get_now(ErlDrvNowData *now)

This function reads a timestamp into the memory pointed to by the parameter `now`. See the description of *ErlDrvNowData* for specification of it's fields.

The return value is 0 unless the `now` pointer is not valid, in which case it is < 0.

int driver_select(ErlDrvPort port, ErlDrvEvent event, int mode, int on)

This function is used by drivers to provide the emulator with events to check for. This enables the emulator to call the driver when something has happened asynchronously.

The `event` argument identifies an OS-specific event object. On Unix systems, the functions `select`/`poll` are used. The event object must be a socket or pipe (or other object that `select`/`poll` can use). On windows, the Win32 API function `WaitForMultipleObjects` is used. This places other restriction on the event object. Refer to the Win32 SDK documentation.

The `on` parameter should be `1` for setting events and `0` for clearing them.

The `mode` argument is bitwise-or combination of `ERL_DRV_READ`, `ERL_DRV_WRITE` and `ERL_DRV_USE`. The first two specifies whether to wait for read events and/or write events. A fired read event will call *ready_input* while a fired write event will call *ready_output*.

> **Note:**
>
> Some OS (Windows) does not differ between read and write events. The call-back for a fired event then only depends on the value of `mode`.

`ERL_DRV_USE` specifies if we are using the event object or if we want to close it. On an emulator with SMP support, it is not safe to clear all events and then close the event object after `driver_select` has returned. Another thread may still be using the event object internally. To safely close an event object call `driver_select` with `ERL_DRV_USE` and `on==0`. That will clear all events and then call *stop_select* when it is safe to close the event object. `ERL_DRV_USE` should be set together with the first event for an event object. It is harmless to set `ERL_DRV_USE` even though it

already has been done. Clearing all events but keeping `ERL_DRV_USE` set will indicate that we are using the event object and probably will set events for it again.

> Note:
>
> ERL_DRV_USE was added in OTP release R13. Old drivers will still work as before. But it is recommended to update them to use `ERL_DRV_USE` and `stop_select` to make sure that event objects are closed in a safe way.

The return value is 0 (Failure, -1, only if the `ready_input`/`ready_output` is `NULL`.

`void * driver_alloc(size_t size)`

This function allocates a memory block of the size specified in `size`, and returns it. This only fails on out of memory, in that case `NULL` is returned. (This is most often a wrapper for `malloc`).

Memory allocated must be explicitly freed with a corresponding call to `driver_free` (unless otherwise stated).

This function is thread-safe.

`void * driver_realloc(void *ptr, size_t size)`

This function resizes a memory block, either in place, or by allocating a new block, copying the data and freeing the old block. A pointer is returned to the reallocated memory. On failure (out of memory), `NULL` is returned. (This is most often a wrapper for `realloc`.)

This function is thread-safe.

`void driver_free(void *ptr)`

This function frees the memory pointed to by `ptr`. The memory should have been allocated with `driver_alloc`. All allocated memory should be deallocated, just once. There is no garbage collection in drivers.

This function is thread-safe.

`ErlDrvBinary* driver_alloc_binary(int size)`

This function allocates a driver binary with a memory block of at least `size` bytes, and returns a pointer to it, or NULL on failure (out of memory). When a driver binary has been sent to the emulator, it must not be altered. Every allocated binary should be freed by a corresponding call to `driver_free_binary` (unless otherwise stated).

Note that a driver binary has an internal reference counter, this means that calling `driver_free_binary` it may not actually dispose of it. If it's sent to the emulator, it may be referenced there.

The driver binary has a field, `orig_bytes`, which marks the start of the data in the binary.

This function is thread-safe.

`ErlDrvBinary* driver_realloc_binary(ErlDrvBinary *bin, int size)`

This function resizes a driver binary, while keeping the data. The resized driver binary is returned. On failure (out of memory), `NULL` is returned.

This function is only thread-safe when the emulator with SMP support is used.

void driver_free_binary(ErlDrvBinary *bin)

This function frees a driver binary `bin`, allocated previously with `driver_alloc_binary`. Since binaries in Erlang are reference counted, the binary may still be around.

This function is only thread-safe when the emulator with SMP support is used.

long driver_binary_get_refc(ErlDrvBinary *bin)

Returns current reference count on `bin`.

This function is only thread-safe when the emulator with SMP support is used.

long driver_binary_inc_refc(ErlDrvBinary *bin)

Increments the reference count on `bin` and returns the reference count reached after the increment.

This function is only thread-safe when the emulator with SMP support is used.

long driver_binary_dec_refc(ErlDrvBinary *bin)

Decrements the reference count on `bin` and returns the reference count reached after the decrement.

This function is only thread-safe when the emulator with SMP support is used.

> Note:
>
> You should normally decrement the reference count of a driver binary by calling *driver_free_binary()*. `driver_binary_dec_refc()` does *not* free the binary if the reference count reaches zero. *Only* use `driver_binary_dec_refc()` when you are sure *not* to reach a reference count of zero.

int driver_enq(ErlDrvPort port, char* buf, int len)

This function enqueues data in the driver queue. The data in `buf` is copied (`len` bytes) and placed at the end of the driver queue. The driver queue is normally used in a FIFO way.

The driver queue is available to queue output from the emulator to the driver (data from the driver to the emulator is queued by the emulator in normal erlang message queues). This can be useful if the driver has to wait for slow devices etc, and wants to yield back to the emulator. The driver queue is implemented as an ErlIOVec.

When the queue contains data, the driver won't close, until the queue is empty.

The return value is 0.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_pushq(ErlDrvPort port, char* buf, int len)

This function puts data at the head of the driver queue. The data in `buf` is copied (`len` bytes) and placed at the beginning of the queue.

The return value is 0.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_deq(ErlDrvPort port, int size)

This function dequeues data by moving the head pointer forward in the driver queue by `size` bytes. The data in the queue will be deallocated.

The return value is the number of bytes remaining in the queue or -1 on failure.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_sizeq(ErlDrvPort port)

This function returns the number of bytes currently in the driver queue.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_enq_bin(ErlDrvPort port, ErlDrvBinary *bin, int offset, int len)

This function enqueues a driver binary in the driver queue. The data in `bin` at `offset` with length `len` is placed at the end of the queue. This function is most often faster than `driver_enq`, because the data doesn't have to be copied.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

The return value is 0.

int driver_pushq_bin(ErlDrvPort port, ErlDrvBinary *bin, int offset, int len)

This function puts data in the binary `bin`, at `offset` with length `len` at the head of the driver queue. It is most often faster than `driver_pushq`, because the data doesn't have to be copied.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

The return value is 0.

SysIOVec* driver_peekq(ErlDrvPort port, int *vlen)

This function retrieves the driver queue as a pointer to an array of `SysIOVec`s. It also returns the number of elements in `vlen`. This is the only way to get data out of the queue.

Nothing is remove from the queue by this function, that must be done with `driver_deq`.

The returned array is suitable to use with the Unix system call `writev`.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_enqv(ErlDrvPort port, ErlIOVec *ev, int skip)

This function enqueues the data in `ev`, skipping the first `skip` bytes of it, at the end of the driver queue. It is faster than `driver_enq`, because the data doesn't have to be copied.

The return value is 0.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

int driver_pushqv(ErlDrvPort port, ErlIOVec *ev, int skip)

This function puts the data in `ev`, skipping the first `skip` bytes of it, at the head of the driver queue. It is faster than `driver_pushq`, because the data doesn't have to be copied.

The return value is 0.

This function can be called from an arbitrary thread if a *port data lock* associated with the `port` is locked by the calling thread during the call.

ErlDrvPDL driver_pdl_create(ErlDrvPort port)

This function creates a port data lock associated with the `port`. *NOTE*: Once a port data lock has been created, it has to be locked during all operations on the driver queue of the `port`.

On success a newly created port data lock is returned. On failure `NULL` is returned. `driver_pdl_create()` will fail if `port` is invalid or if a port data lock already has been associated with the `port`.

void driver_pdl_lock(ErlDrvPDL pdl)

This function locks the port data lock passed as argument (`pdl`).

This function is thread-safe.

void driver_pdl_unlock(ErlDrvPDL pdl)

This function unlocks the port data lock passed as argument (`pdl`).

This function is thread-safe.

long driver_pdl_get_refc(ErlDrvPDL pdl)

This function returns the current reference count of the port data lock passed as argument (`pdl`).

This function is thread-safe.

long driver_pdl_inc_refc(ErlDrvPDL pdl)

This function increments the reference count of the port data lock passed as argument (`pdl`).

The current reference count after the increment has been performed is returned.

This function is thread-safe.

long driver_pdl_dec_refc(ErlDrvPDL pdl)

This function decrements the reference count of the port data lock passed as argument (`pdl`).

The current reference count after the decrement has been performed is returned.

This function is thread-safe.

int driver_monitor_process(ErlDrvPort port, ErlDrvTermData process, ErlDrvMonitor *monitor)

Start monitoring a process from a driver. When a process is monitored, a process exit will result in a call to the provided *process_exit* call-back in the *ErlDrvEntry* structure. The `ErlDrvMonitor` structure is filled in, for later removal or compare.

The `process` parameter should be the return value of an earlier call to *driver_caller* or *driver_connected* call.

The function returns 0 on success, < 0 if no call-back is provided and > 0 if the process is no longer alive.

int driver_demonitor_process(ErlDrvPort port, const ErlDrvMonitor *monitor)

This function cancels an monitor created earlier.

The function returns 0 if a monitor was removed and > 0 if the monitor did no longer exist.

ErlDrvTermData driver_get_monitored_process(ErlDrvPort port, const ErlDrvMonitor *monitor)

The function returns the process id associated with a living monitor. It can be used in the `process_exit` call-back to get the process identification for the exiting process.

The function returns `driver_term_nil` if the monitor no longer exists.

int driver_compare_monitors(const ErlDrvMonitor *monitor1, const ErlDrvMonitor *monitor2)

This function is used to compare two `ErlDrvMonitor`s. It can also be used to imply some artificial order on monitors, for whatever reason.

The function returns 0 if `monitor1` and `monitor2` are equal, < 0 if `monitor1` is less than `monitor2` and > 0 if `monitor1` is greater than `monitor2`.

void add_driver_entry(ErlDrvEntry *de)

This function adds a driver entry to the list of drivers known by Erlang. The *init* function of the `de` parameter is called.

> Note:
>
> To use this function for adding drivers residing in dynamically loaded code is dangerous. If the driver code for the added driver resides in the same dynamically loaded module (i.e. `.so` file) as a normal dynamically loaded driver (loaded with the `erl_ddll` interface), the caller should call *driver_lock_driver* before adding driver entries.
>
> Use of this function is generally deprecated.

int remove_driver_entry(ErlDrvEntry *de)

This function removes a driver entry `de` previously added with `add_driver_entry`.

Driver entries added by the `erl_ddll` erlang interface can not be removed by using this interface.

char* erl_errno_id(int error)

This function returns the atom name of the erlang error, given the error number in `error`. Error atoms are: `einval`, `enoent`, etc. It can be used to make error terms from the driver.

void set_busy_port(ErlDrvPort port, int on)

This function set and resets the busy status of the port. If `on` is 1, the port is set to busy, if it's 0 the port is set to not busy.

When the port is busy, sending to it with `Port ! Data` or `port_command/2`, will block the port owner process, until the port is signaled as not busy.

If the *ERL_DRV_FLAG_SOFT_BUSY* has been set in the *driver_entry*, data can be forced into the driver via *port_command(Port, Data, [force])* even though the driver has signaled that it is busy.

void set_port_control_flags(ErlDrvPort port, int flags)

This function sets flags for how the *control* driver entry function will return data to the port owner process. (The `control` function is called from `port_control/3` in erlang.)

Currently there are only two meaningful values for `flags`: 0 means that data is returned in a list, and `PORT_CONTROL_FLAG_BINARY` means data is returned as a binary from `control`.

int driver_failure_eof(ErlDrvPort port)

This function signals to erlang that the driver has encountered an EOF and should be closed, unless the port was opened with the `eof` option, in that case eof is sent to the port. Otherwise, the port is close and an `'EXIT'` message is sent to the port owner process.

The return value is 0.

int driver_failure_atom(ErlDrvPort port, char *string)
int driver_failure_posix(ErlDrvPort port, int error)
int driver_failure(ErlDrvPort port, int error)

These functions signal to Erlang that the driver has encountered an error and should be closed. The port is closed and the tuple `{'EXIT', error, Err}`, is sent to the port owner process, where error is an error atom (`driver_failure_atom` and `driver_failure_posix`), or an integer (`driver_failure`).

The driver should fail only when in severe error situations, when the driver cannot possibly keep open, for instance buffer allocation gets out of memory. Normal errors is more appropriate to handle with sending error codes with `driver_output`.

The return value is 0.

ErlDrvTermData driver_connected(ErlDrvPort port)

This function returns the port owner process.

ErlDrvTermData driver_caller(ErlDrvPort port)

This function returns the process id of the process that made the current call to the driver. The process id can be used with `driver_send_term` to send back data to the caller. `driver_caller()` only return valid data when currently executing in one of the following driver callbacks:

start
: Called from `open_port/2`.

output
: Called from `erlang:send/2`, and `erlang:port_command/2`

outputv
: Called from `erlang:send/2`, and `erlang:port_command/2`

control
: Called from `erlang:port_control/3`

call
: Called from `erlang:port_call/3`

int driver_output_term(ErlDrvPort port, ErlDrvTermData* term, int n)

This functions sends data in the special driver term format. This is a fast way to deliver term data from a driver. It also needs no binary conversion, so the port owner process receives data as normal Erlang terms.

The `term` parameter points to an array of `ErlDrvTermData`, with `n` elements. This array contains terms described in the driver term format. Every term consists of one to four elements in the array. The term first has a term type, and then arguments.

Tuple and lists (with the exception of strings, see below), are built in reverse polish notation, so that to build a tuple, the elements are given first, and then the tuple term, with a count. Likewise for lists.

A tuple must be specified with the number of elements. (The elements precedes the `ERL_DRV_TUPLE` term.)

A list must be specified with the number of elements, including the tail, which is the last term preceding `ERL_DRV_LIST`.

The special term `ERL_DRV_STRING_CONS` is used to "splice" in a string in a list, a string given this way is not a list per se, but the elements are elements of the surrounding list.

```
Term type            Argument(s)
========================================
ERL_DRV_NIL
ERL_DRV_ATOM         ErlDrvTermData atom (from driver_mk_atom(char *string))
ERL_DRV_INT          ErlDrvSInt integer
ERL_DRV_UINT         ErlDrvUInt integer
ERL_DRV_INT64        ErlDrvSInt64 *integer_ptr
ERL_DRV_UINT64       ErlDrvUInt64 *integer_ptr
ERL_DRV_PORT         ErlDrvTermData port (from driver_mk_port(ErlDrvPort port))
ERL_DRV_BINARY       ErlDrvBinary *bin, ErlDrvUInt len, ErlDrvUInt offset
ERL_DRV_BUF2BINARY   char *buf, ErlDrvUInt len
ERL_DRV_STRING       char *str, int len
ERL_DRV_TUPLE        int sz
ERL_DRV_LIST         int sz
ERL_DRV_PID          ErlDrvTermData pid (from driver_connected(ErlDrvPort port) or driver_caller(ErlDrvPort port
ERL_DRV_STRING_CONS  char *str, int len
ERL_DRV_FLOAT        double *dbl
ERL_DRV_EXT2TERM     char *buf, ErlDrvUInt len
```

The unsigned integer data type `ErlDrvUInt` and the signed integer data type `ErlDrvSInt` are 64 bits wide on a 64 bit runtime system and 32 bits wide on a 32 bit runtime system. They were introduced in erts version 5.6, and replaced some of the `int` arguments in the list above.

The unsigned integer data type `ErlDrvUInt64` and the signed integer data type `ErlDrvSInt64` are always 64 bits wide. They were introduced in erts version 5.7.4.

To build the tuple `{tcp, Port, [100 | Binary]}`, the following call could be made.

```
ErlDrvBinary* bin = ...
ErlDrvPort port = ...
ErlDrvTermData spec[] = {
    ERL_DRV_ATOM, driver_mk_atom("tcp"),
    ERL_DRV_PORT, driver_mk_port(port),
        ERL_DRV_INT, 100,
        ERL_DRV_BINARY, bin, 50, 0,
        ERL_DRV_LIST, 2,
    ERL_DRV_TUPLE, 3,
};
driver_output_term(port, spec, sizeof(spec) / sizeof(spec[0]));
```

Where bin is a driver binary of length at least 50 and port is a port handle. Note that the ERL_DRV_LIST comes after the elements of the list, likewise the ERL_DRV_TUPLE.

The term ERL_DRV_STRING_CONS is a way to construct strings. It works differently from how ERL_DRV_STRING works. ERL_DRV_STRING_CONS builds a string list in reverse order, (as opposed to how ERL_DRV_LIST works), concatenating the strings added to a list. The tail must be given before ERL_DRV_STRING_CONS.

The ERL_DRV_STRING constructs a string, and ends it. (So it's the same as ERL_DRV_NIL followed by ERL_DRV_STRING_CONS.)

```
/* to send [x, "abc", y] to the port: */
ErlDrvTermData spec[] = {
    ERL_DRV_ATOM, driver_mk_atom("x"),
    ERL_DRV_STRING, (ErlDrvTermData)"abc", 3,
    ERL_DRV_ATOM, driver_mk_atom("y"),
    ERL_DRV_NIL,
    ERL_DRV_LIST, 4
};
driver_output_term(port, spec, sizeof(spec) / sizeof(spec[0]));
```

```
/* to send "abc123" to the port: */
ErlDrvTermData spec[] = {
    ERL_DRV_NIL,        /* with STRING_CONS, the tail comes first */
    ERL_DRV_STRING_CONS, (ErlDrvTermData)"123", 3,
    ERL_DRV_STRING_CONS, (ErlDrvTermData)"abc", 3,
};
driver_output_term(port, spec, sizeof(spec) / sizeof(spec[0]));
```

The ERL_DRV_EXT2TERM term type is used for passing a term encoded with the *external format*, i.e., a term that has been encoded by *erlang:term_to_binary*, *erl_interface*, etc. For example, if binp is a pointer to an ErlDrvBinary that contains the term {17, 4711} encoded with the *external format* and you want to wrap it in a two tuple with the tag my_tag, i.e., {my_tag, {17, 4711}}, you can do as follows:

```
ErlDrvTermData spec[] = {
        ERL_DRV_ATOM, driver_mk_atom("my_tag"),
        ERL_DRV_EXT2TERM, (ErlDrvTermData) binp->orig_bytes, binp->orig_size
    ERL_DRV_TUPLE, 2,
};
driver_output_term(port, spec, sizeof(spec) / sizeof(spec[0]));
```

If you want to pass a binary and doesn't already have the content of the binary in an ErlDrvBinary, you can benefit from using ERL_DRV_BUF2BINARY instead of creating an ErlDrvBinary via driver_alloc_binary() and then pass the binary via ERL_DRV_BINARY. The runtime system will often allocate binaries smarter if ERL_DRV_BUF2BINARY is used. However, if the content of the binary to pass already resides in an ErlDrvBinary, it is normally better to pass the binary using ERL_DRV_BINARY and the ErlDrvBinary in question.

The ERL_DRV_UINT, ERL_DRV_BUF2BINARY, and ERL_DRV_EXT2TERM term types were introduced in the 5.6 version of erts.

Note that this function is *not* thread-safe, not even when the emulator with SMP support is used.

ErlDrvTermData driver_mk_atom(char* string)

This function returns an atom given a name `string`. The atom is created and won't change, so the return value may be saved and reused, which is faster than looking up the atom several times.

ErlDrvTermData driver_mk_port(ErlDrvPort port)

This function converts a port handle to the erlang term format, usable in the `driver_output_send` function.

int driver_send_term(ErlDrvPort port, ErlDrvTermData receiver, ErlDrvTermData* term, int n)

This function is the only way for a driver to send data to *other* processes than the port owner process. The `receiver` parameter specifies the process to receive the data.

The parameters `term` and `n` does the same thing as in *driver_output_term*.

This function is only thread-safe when the emulator with SMP support is used.

long driver_async (ErlDrvPort port, unsigned int* key, void (*async_invoke)(void*), void* async_data, void (*async_free)(void*))

This function performs an asynchronous call. The function `async_invoke` is invoked in a thread separate from the emulator thread. This enables the driver to perform time-consuming, blocking operations without blocking the emulator.

Erlang is by default started without an async thread pool. The number of async threads that the runtime system should use is specified by the *+A* command line argument of *erl(1)*. If no async thread pool is available, the call is made synchronously in the thread calling `driver_async()`. The current number of async threads in the async thread pool can be retrieved via *driver_system_info()*.

If there is a thread pool available, a thread will be used. If the `key` argument is null, the threads from the pool are used in a round-robin way, each call to `driver_async` uses the next thread in the pool. With the `key` argument set, this behaviour is changed. The two same values of `*key` always get the same thread.

To make sure that a driver instance always uses the same thread, the following call can be used:

```
unsigned int myKey = (unsigned int) myPort;

r = driver_async(myPort, &myKey, myData, myFunc);
```

It is enough to initialize `myKey` once for each driver instance.

If a thread is already working, the calls will be queued up and executed in order. Using the same thread for each driver instance ensures that the calls will be made in sequence.

The `async_data` is the argument to the functions `async_invoke` and `async_free`. It's typically a pointer to a structure that contains a pipe or event that can be used to signal that the async operation completed. The data should be freed in `async_free`, because it's called if `driver_async_cancel` is called.

When the async operation is done, *ready_async* driver entry function is called. If `async_ready` is null in the driver entry, the `async_free` function is called instead.

The return value is a handle to the asynchronous task, which can be used as argument to `driver_async_cancel`.

> Note:
>
> As of erts version 5.5.4.3 the default stack size for threads in the async-thread pool is 16 kilowords, i.e., 64 kilobyte on 32-bit architectures. This small default size has been chosen since the amount of async-threads might be quite large. The default stack size is enough for drivers delivered with Erlang/OTP, but might not be sufficiently large for other dynamically linked in drivers that use the driver_async() functionality. A suggested stack size for threads in the async-thread pool can be configured via the *+a* command line argument of *erl(1)*.

`int driver_async_cancel(long id)`

This function cancels an asynchronous operation, by removing it from the queue. Only functions in the queue can be cancelled; if a function is executing, it's too late to cancel it. The `async_free` function is also called.

The return value is 1 if the operation was removed from the queue, otherwise 0.

`int driver_lock_driver(ErlDrvPort port)`

This function locks the driver used by the port `port` in memory for the rest of the emulator process lifetime. After this call, the driver behaves as one of Erlang's statically linked in drivers.

`ErlDrvPort driver_create_port(ErlDrvPort port, ErlDrvTermData owner_pid, char* name, ErlDrvData drv_data)`

This function creates a new port executing the same driver code as the port creating the new port. A short description of the arguments:

`port`
: The port handle of the port (driver instance) creating the new port.

`owner_pid`
: The process id of the Erlang process which will be owner of the new port. This process will be linked to the new port. You usually want to use `driver_caller(port)` as `owner_pid`.

`name`
: The port name of the new port. You usually want to use the same port name as the driver name (*driver_name* field of the *driver_entry*).

`drv_data`
: The driver defined handle that will be passed in subsequent calls to driver call-backs. Note, that the *driver start call-back* will not be called for this new driver instance. The driver defined handle is normally created in the *driver start call-back* when a port is created via *erlang:open_port/2*.

The caller of `driver_create_port()` is allowed to manipulate the newly created port when `driver_create_port()` has returned. When *port level locking* is used, the creating port is, however, only allowed to manipulate the newly created port until the current driver call-back that was called by the emulator returns.

> Note:
>
> When *port level locking* is used, the creating port is only allowed to manipulate the newly created port until the current driver call-back returns.

`int erl_drv_thread_create(char *name, ErlDrvTid *tid, void * (*func)(void *), void *arg, ErlDrvThreadOpts *opts)`

Arguments:

`name`
: A string identifying the created thread. It will be used to identify the thread in planned future debug functionality.

`tid`
: A pointer to a thread identifier variable.

`func`
: A pointer to a function to execute in the created thread.

`arg`
: A pointer to argument to the `func` function.

`opts`
: A pointer to thread options to use or `NULL`.

This function creates a new thread. On success `0` is returned; otherwise, an `errno` value is returned to indicate the error. The newly created thread will begin executing in the function pointed to by `func`, and `func` will be passed `arg` as argument. When `erl_drv_thread_create()` returns the thread identifier of the newly created thread will be available in `*tid`. `opts` can be either a `NULL` pointer, or a pointer to an *ErlDrvThreadOpts* structure. If `opts` is a `NULL` pointer, default options will be used; otherwise, the passed options will be used.

> ### Warning:
>
> You are not allowed to allocate the *ErlDrvThreadOpts* structure by yourself. It has to be allocated and initialized by *erl_drv_thread_opts_create()*.

The created thread will terminate either when `func` returns or if *erl_drv_thread_exit()* is called by the thread. The exit value of the thread is either returned from `func` or passed as argument to *erl_drv_thread_exit()*. The driver creating the thread has the responsibility of joining the thread, via *erl_drv_thread_join()*, before the driver is unloaded. It is not possible to create "detached" threads, i.e., threads that don't need to be joined.

> ### Warning:
>
> All created threads need to be joined by the driver before it is unloaded. If the driver fails to join all threads created before it is unloaded, the runtime system will most likely crash when the code of the driver is unloaded.

This function is thread-safe.

`ErlDrvThreadOpts * erl_drv_thread_opts_create(char *name)`

Arguments:

`name`
: A string identifying the created thread options. It will be used to identify the thread options in planned future debug functionality.

This function allocates and initialize a thread option structure. On failure `NULL` is returned. A thread option structure is used for passing options to *erl_drv_thread_create()*. If the structure isn't modified before it is passed to *erl_drv_thread_create()*, the default values will be used.

> Warning:
>
> You are not allowed to allocate the *ErlDrvThreadOpts* structure by yourself. It has to be allocated and initialized by `erl_drv_thread_opts_create()`.

This function is thread-safe.

`void erl_drv_thread_opts_destroy(ErlDrvThreadOpts *opts)`

Arguments:

`opts`
: A pointer to thread options to destroy.

This function destroys thread options previously created by *erl_drv_thread_opts_create()*.

This function is thread-safe.

`void erl_drv_thread_exit(void *exit_value)`

Arguments:

`exit_value`
: A pointer to an exit value or `NULL`.

This function terminates the calling thread with the exit value passed as argument. You are only allowed to terminate threads created with *erl_drv_thread_create()*. The exit value can later be retrieved by another thread via *erl_drv_thread_join()*.

This function is thread-safe.

`int erl_drv_thread_join(ErlDrvTid tid, void **exit_value)`

Arguments:

`tid`
: The thread identifier of the thread to join.

`exit_value`
: A pointer to a pointer to an exit value, or `NULL`.

This function joins the calling thread with another thread, i.e., the calling thread is blocked until the thread identified by `tid` has terminated. On success `0` is returned; otherwise, an `errno` value is returned to indicate the error. A thread can only be joined once. The behavior of joining more than once is undefined, an emulator crash is likely. If `exit_value == NULL`, the exit value of the terminated thread will be ignored; otherwise, the exit value of the terminated thread will be stored at `*exit_value`.

This function is thread-safe.

`ErlDrvTid erl_drv_thread_self(void)`

This function returns the thread identifier of the calling thread.

This function is thread-safe.

`int erl_drv_equal_tids(ErlDrvTid tid1, ErlDrvTid tid2)`

Arguments:

`tid1`
A thread identifier.
`tid2`
A thread identifier.

This function compares two thread identifiers for equality, and returns `0` it they aren't equal, and a value not equal to `0` if they are equal.

> ### Note:
>
> A Thread identifier may be reused very quickly after a thread has terminated. Therefore, if a thread corresponding to one of the involved thread identifiers has terminated since the thread identifier was saved, the result of `erl_drv_equal_tids()` might not give expected result.

This function is thread-safe.

`ErlDrvMutex * erl_drv_mutex_create(char *name)`

Arguments:

`name`
A string identifying the created mutex. It will be used to identify the mutex in planned future debug functionality.

This function creates a mutex and returns a pointer to it. On failure `NULL` is returned. The driver creating the mutex has the responsibility of destroying it before the driver is unloaded.

This function is thread-safe.

`void erl_drv_mutex_destroy(ErlDrvMutex *mtx)`

Arguments:

`mtx`
A pointer to a mutex to destroy.

This function destroys a mutex previously created by *erl_drv_mutex_create()*. The mutex has to be in an unlocked state before being destroyed.

This function is thread-safe.

`void erl_drv_mutex_lock(ErlDrvMutex *mtx)`

Arguments:

`mtx`
A pointer to a mutex to lock.

This function locks a mutex. The calling thread will be blocked until the mutex has been locked. A thread which currently has locked the mutex may *not* lock the same mutex again.

> ### Warning:
>
> If you leave a mutex locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

`int erl_drv_mutex_trylock(ErlDrvMutex *mtx)`

Arguments:

`mtx`
: A pointer to a mutex to try to lock.

This function tries to lock a mutex. If successful `0`, is returned; otherwise, `EBUSY` is returned. A thread which currently has locked the mutex may *not* try to lock the same mutex again.

> Warning:
>
> If you leave a mutex locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

`void erl_drv_mutex_unlock(ErlDrvMutex *mtx)`

Arguments:

`mtx`
: A pointer to a mutex to unlock.

This function unlocks a mutex. The mutex currently has to be locked by the calling thread.

This function is thread-safe.

`ErlDrvCond * erl_drv_cond_create(char *name)`

Arguments:

`name`
: A string identifying the created condition variable. It will be used to identify the condition variable in planned future debug functionality.

This function creates a condition variable and returns a pointer to it. On failure `NULL` is returned. The driver creating the condition variable has the responsibility of destroying it before the driver is unloaded.

This function is thread-safe.

`void erl_drv_cond_destroy(ErlDrvCond *cnd)`

Arguments:

`cnd`
: A pointer to a condition variable to destroy.

This function destroys a condition variable previously created by *erl_drv_cond_create()*.

This function is thread-safe.

`void erl_drv_cond_signal(ErlDrvCond *cnd)`

Arguments:

`cnd`
A pointer to a condition variable to signal on.

This function signals on a condition variable. That is, if other threads are waiting on the condition variable being signaled, *one* of them will be woken.

This function is thread-safe.

`void erl_drv_cond_broadcast(ErlDrvCond *cnd)`

Arguments:

`cnd`
A pointer to a condition variable to broadcast on.

This function broadcasts on a condition variable. That is, if other threads are waiting on the condition variable being broadcasted on, *all* of them will be woken.

This function is thread-safe.

`void erl_drv_cond_wait(ErlDrvCond *cnd, ErlDrvMutex *mtx)`

Arguments:

`cnd`
A pointer to a condition variable to wait on.

`mtx`
A pointer to a mutex to unlock while waiting.

This function waits on a condition variable. The calling thread is blocked until another thread wakes it by signaling or broadcasting on the condition variable. Before the calling thread is blocked it unlocks the mutex passed as argument, and when the calling thread is woken it locks the same mutex before returning. That is, the mutex currently has to be locked by the calling thread when calling this function.

> **Note:**
>
> `erl_drv_cond_wait()` might return even though no-one has signaled or broadcasted on the condition variable. Code calling `erl_drv_cond_wait()` should always be prepared for `erl_drv_cond_wait()` returning even though the condition that the thread was waiting for hasn't occurred. That is, when returning from `erl_drv_cond_wait()` always check if the condition has occurred, and if not call `erl_drv_cond_wait()` again.

This function is thread-safe.

`ErlDrvRWLock * erl_drv_rwlock_create(char *name)`

Arguments:

`name`
A string identifying the created rwlock. It will be used to identify the rwlock in planned future debug functionality.

This function creates an rwlock and returns a pointer to it. On failure `NULL` is returned. The driver creating the rwlock has the responsibility of destroying it before the driver is unloaded.

This function is thread-safe.

`void erl_drv_rwlock_destroy(ErlDrvRWLock *rwlck)`

Arguments:

`rwlck`
: A pointer to an rwlock to destroy.

This function destroys an rwlock previously created by *erl_drv_rwlock_create()*. The rwlock has to be in an unlocked state before being destroyed.

This function is thread-safe.

`void erl_drv_rwlock_rlock(ErlDrvRWLock *rwlck)`

Arguments:

`rwlck`
: A pointer to an rwlock to read lock.

This function read locks an rwlock. The calling thread will be blocked until the rwlock has been read locked. A thread which currently has read or read/write locked the rwlock may *not* lock the same rwlock again.

> Warning:
>
> If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

`int erl_drv_rwlock_tryrlock(ErlDrvRWLock *rwlck)`

Arguments:

`rwlck`
: A pointer to an rwlock to try to read lock.

This function tries to read lock an rwlock. If successful `0`, is returned; otherwise, `EBUSY` is returned. A thread which currently has read or read/write locked the rwlock may *not* try to lock the same rwlock again.

> Warning:
>
> If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

`void erl_drv_rwlock_runlock(ErlDrvRWLock *rwlck)`

Arguments:

`rwlck`
: A pointer to an rwlock to read unlock.

This function read unlocks an rwlock. The rwlock currently has to be read locked by the calling thread.

This function is thread-safe.

void erl_drv_rwlock_rwlock(ErlDrvRWLock *rwlck)

Arguments:

`rwlck`
: A pointer to an rwlock to read/write lock.

This function read/write locks an rwlock. The calling thread will be blocked until the rwlock has been read/write locked. A thread which currently has read or read/write locked the rwlock may *not* lock the same rwlock again.

> Warning:
>
> If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

int erl_drv_rwlock_tryrwlock(ErlDrvRWLock *rwlck)

Arguments:

`rwlck`
: A pointer to an rwlock to try to read/write lock.

This function tries to read/write lock an rwlock. If successful `0`, is returned; otherwise, `EBUSY` is returned. A thread which currently has read or read/write locked the rwlock may *not* try to lock the same rwlock again.

> Warning:
>
> If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will *very likely* deadlock the whole emulator.

This function is thread-safe.

void erl_drv_rwlock_rwunlock(ErlDrvRWLock *rwlck)

Arguments:

`rwlck`
: A pointer to an rwlock to read/write unlock.

This function read/write unlocks an rwlock. The rwlock currently has to be read/write locked by the calling thread.

This function is thread-safe.

int erl_drv_tsd_key_create(char *name, ErlDrvTSDKey *key)

Arguments:

`name`
: A string identifying the created key. It will be used to identify the key in planned future debug functionality.

`key`
: A pointer to a thread specific data key variable.

This function creates a thread specific data key. On success `0` is returned; otherwise, an `errno` value is returned to indicate the error. The driver creating the key has the responsibility of destroying it before the driver is unloaded.

This function is thread-safe.

`void erl_drv_tsd_key_destroy(ErlDrvTSDKey key)`

Arguments:

`key`
: A thread specific data key to destroy.

This function destroys a thread specific data key previously created by *erl_drv_tsd_key_create()*. All thread specific data using this key in all threads have to be cleared (see *erl_drv_tsd_set()*) prior to the call to `erl_drv_tsd_key_destroy()`.

> ## Warning:
>
> A destroyed key is very likely to be reused soon. Therefore, if you fail to clear the thread specific data using this key in a thread prior to destroying the key, you will *very likely* get unexpected errors in other parts of the system.

This function is thread-safe.

`void erl_drv_tsd_set(ErlDrvTSDKey key, void *data)`

Arguments:

`key`
: A thread specific data key.

`data`
: A pointer to data to associate with `key` in calling thread.

This function sets thread specific data associated with `key` for the calling thread. You are only allowed to set thread specific data for threads while they are fully under your control. For example, if you set thread specific data in a thread calling a driver call-back function, it has to be cleared, i.e. set to `NULL`, before returning from the driver call-back function.

> ## Warning:
>
> If you fail to clear thread specific data in an emulator thread before letting it out of your control, you might not ever be able to clear this data with later unexpected errors in other parts of the system as a result.

This function is thread-safe.

`void * erl_drv_tsd_get(ErlDrvTSDKey key)`

Arguments:

`key`
: A thread specific data key.

This function returns the thread specific data associated with `key` for the calling thread. If no data has been associated with `key` for the calling thread, `NULL` is returned.

This function is thread-safe.

int erl_drv_putenv(char *key, char *value)

Arguments:

`key`
: A null terminated string containing the name of the environment variable.

`value`
: A null terminated string containing the new value of the environment variable.

This function sets the value of an environment variable. It returns `0` on success, and a value `!= 0` on failure.

> **Note:**
>
> The result of passing the empty string ("") as a value is platform dependent. On some platforms the value of the variable is set to the empty string, on others, the environment variable is removed.

> **Warning:**
>
> Do *not* use libc's `putenv` or similar C library interfaces from a driver.

This function is thread-safe.

int erl_drv_getenv(char *key, char *value, size_t *value_size)

Arguments:

`key`
: A null terminated string containing the name of the environment variable.

`value`
: A pointer to an output buffer.

`value_size`
: A pointer to an integer. The integer is both used for passing input and output sizes (see below).

This function retrieves the value of an environment variable. When called, `*value_size` should contain the size of the `value` buffer. On success `0` is returned, the value of the environment variable has been written to the `value` buffer, and `*value_size` contains the string length (excluding the terminating null character) of the value written to the `value` buffer. On failure, i.e., no such environment variable was found, a value less than `0` is returned. When the size of the `value` buffer is too small, a value greater than `0` is returned and `*value_size` has been set to the buffer size needed.

> **Warning:**
>
> Do *not* use libc's `getenv` or similar C library interfaces from a driver.

This function is thread-safe.

SEE ALSO

driver_entry(3), *erl_ddll(3)*, *erlang(3)*

An Alternative Distribution Driver (ERTS User's Guide Ch. 3)

driver_entry

C Library

As of erts version 5.5.3 the driver interface has been extended (see *extended marker*). The extended interface introduce *version management*, the possibility to pass capability flags (see *driver flags*) to the runtime system at driver initialization, and some new driver API functions.

> ### Note:
>
> Old drivers (compiled with an `erl_driver.h` from an earlier erts version than 5.5.3) have to be recompiled (but does not have to use the extended interface).

The `driver_entry` structure is a C struct that all erlang drivers defines. It contains entry points for the erlang driver that are called by the erlang emulator when erlang code accesses the driver.

The *erl_driver* driver API functions needs a port handle that identifies the driver instance (and the port in the emulator). This is only passed to the `start` function, but not to the other functions. The `start` function returns a driver-defined handle that is passed to the other functions. A common practice is to have the `start` function allocating some application-defined structure and stash the `port` handle in it, to use it later with the driver API functions.

The driver call-back functions are called synchronously from the erlang emulator. If they take too long before completing, they can cause timeouts in the emulator. Use the queue or asynchronous calls if necessary, since the emulator must be responsive.

The driver structure contains the name of the driver and some 15 function pointers. These pointers are called at different times by the emulator.

The only exported function from the driver is `driver_init`. This function returns the `driver_entry` structure that points to the other functions in the driver. The `driver_init` function is declared with a macro `DRIVER_INIT(drivername)`. (This is because different OS's have different names for it.)

When writing a driver in C++, the driver entry should be of `"C"` linkage. One way to do this is to put this line somewhere before the driver entry: `extern "C" DRIVER_INIT(drivername);`.

When the driver has passed the `driver_entry` over to the emulator, the driver is *not* allowed to modify the `driver_entry`.

> ### Note:
>
> Do *not* declare the `driver_entryconst`. This since the emulator needs to modify the `handle`, and the `handle2` fields. A statically allocated, and `const` declared `driver_entry` may be located in read only memory which will cause the emulator to crash.

DATA TYPES

ErlDrvEntry

```
typedef struct erl_drv_entry {
    int (*init)(void);          /* called at system start up for statically
```

```
                                linked drivers, and after loading for
                                dynamically loaded drivers */

#ifndef ERL_SYS_DRV
    ErlDrvData (*start)(ErlDrvPort port, char *command);
                                /* called when open_port/2 is invoked.
                                   return value -1 means failure. */
#else
    ErlDrvData (*start)(ErlDrvPort port, char *command, SysDriverOpts* opts);
                                /* special options, only for system driver */
#endif
    void (*stop)(ErlDrvData drv_data);
                                /* called when port is closed, and when the
                                   emulator is halted. */
    void (*output)(ErlDrvData drv_data, char *buf, int len);
                                /* called when we have output from erlang to
                                   the port */
    void (*ready_input)(ErlDrvData drv_data, ErlDrvEvent event);
                                /* called when we have input from one of
                                   the driver's handles) */
    void (*ready_output)(ErlDrvData drv_data, ErlDrvEvent event);
                                /* called when output is possible to one of
                                   the driver's handles */
    char *driver_name;          /* name supplied as command
                                   in open_port XXX ? */
    void (*finish)(void);       /* called before unloading the driver -
                                   DYNAMIC DRIVERS ONLY */
    void *handle;               /* Reserved -- Used by emulator internally */
    int (*control)(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);
                                /* "ioctl" for drivers - invoked by
                                   port_control/3) */
    void (*timeout)(ErlDrvData drv_data);        /* Handling of timeout in driver */
    void (*outputv)(ErlDrvData drv_data, ErlIOVec *ev);
                                /* called when we have output from erlang
                                   to the port */
    void (*ready_async)(ErlDrvData drv_data, ErlDrvThreadData thread_data);
    void (*flush)(ErlDrvData drv_data);
                                /* called when the port is about to be
                                   closed, and there is data in the
                                   driver queue that needs to be flushed
                                   before 'stop' can be called */
    int (*call)(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen, unsigned int *flags);
                                /* Works mostly like 'control', a syncronous
                                   call into the driver. */
    void (*event)(ErlDrvData drv_data, ErlDrvEvent event,
                  ErlDrvEventData event_data);
                                /* Called when an event selected by
                                   driver_event() has occurred */
    int extended_marker;        /* ERL_DRV_EXTENDED_MARKER */
    int major_version;          /* ERL_DRV_EXTENDED_MAJOR_VERSION */
    int minor_version;          /* ERL_DRV_EXTENDED_MINOR_VERSION */
    int driver_flags;           /* ERL_DRV_FLAGs */
    void *handle2;              /* Reserved -- Used by emulator internally */
    void (*process_exit)(ErlDrvData drv_data, ErlDrvMonitor *monitor);
                                /* Called when a process monitor fires */
    void (*stop_select)(ErlDrvEvent event, void* reserved);
                                /* Called to close an event object */
} ErlDrvEntry;
```

int (*init)(void)

This is called directly after the driver has been loaded by `erl_ddll:load_driver/2`. (Actually when the driver is added to the driver list.) The driver should return 0, or if the driver can't initialize, -1.

int (*start)(ErlDrvPort port, char* command)

This is called when the driver is instantiated, when `open_port/2` is called. The driver should return a number >= 0 or a pointer, or if the driver can't be started, one of three error codes should be returned:

ERL_DRV_ERROR_GENERAL - general error, no error code

ERL_DRV_ERROR_ERRNO - error with error code in erl_errno

ERL_DRV_ERROR_BADARG - error, badarg

If an error code is returned, the port isn't started.

void (*stop)(ErlDrvData drv_data)

This is called when the port is closed, with `port_close/1` or `Port ! {self(), close}`. Note that terminating the port owner process also closes the port.

void (*output)(ErlDrvData drv_data, char *buf, int len)

This is called when an erlang process has sent data to the port. The data is pointed to by `buf`, and is `len` bytes. Data is sent to the port with `Port ! {self(), {command, Data}}`, or with `port_command/2`. Depending on how the port was opened, it should be either a list of integers 0...255 or a binary. See `open_port/3` and `port_command/2`.

void (*ready_input)(ErlDrvData drv_data, ErlDrvEvent event)
void (*ready_output)(ErlDrvData drv_data, ErlDrvEvent event)

This is called when a driver event (given in the `event` parameter) is signaled. This is used to help asynchronous drivers "wake up" when something happens.

On unix the `event` is a pipe or socket handle (or something that the `select` system call understands).

On Windows the `event` is an Event or Semaphore (or something that the `WaitForMultipleObjects` API function understands). (Some trickery in the emulator allows more than the built-in limit of 64 `Events` to be used.)

To use this with threads and asynchronous routines, create a pipe on unix and an Event on Windows. When the routine completes, write to the pipe (use `SetEvent` on Windows), this will make the emulator call `ready_input` or `ready_output`.

char *driver_name

This is the name of the driver, it must correspond to the atom used in `open_port`, and the name of the driver library file (without the extension).

void (*finish)(void)

This function is called by the `erl_ddll` driver when the driver is unloaded. (It is only called in dynamic drivers.)

The driver is only unloaded as a result of calling `unload_driver/1`, or when the emulator halts.

void *handle

This field is reserved for the emulators internal use. The emulator will modify this field; therefore, it is important that the `driver_entry` isn't declared `const`.

int (*control)(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen)

This is a special routine invoked with the erlang function `port_control/3`. It works a little like an "ioctl" for erlang drivers. The data given to `port_control/3` arrives in `buf` and `len`. The driver may send data back, using `*rbuf` and `rlen`.

This is the fastest way of calling a driver and get a response. It won't make any context switch in the erlang emulator, and requires no message passing. It is suitable for calling C function to get faster execution, when erlang is too slow.

If the driver wants to return data, it should return it in `rbuf`. When `control` is called, `*rbuf` points to a default buffer of `rlen` bytes, which can be used to return data. Data is returned different depending on the port control flags (those that are set with *set_port_control_flags*).

If the flag is set to `PORT_CONTROL_FLAG_BINARY`, a binary will be returned. Small binaries can be returned by writing the raw data into the default buffer. A binary can also be returned by setting `*rbuf` to point to a binary allocated with *driver_alloc_binary*. This binary will be freed automatically after `control` has returned. The driver can retain the binary for *read only* access with *driver_binary_inc_refc* to be freed later with *driver_free_binary*. It is never allowed to alter the binary after `control` has returned. If `*rbuf` is set to NULL, an empty list will be returned.

If the flag is set to `0`, data is returned as a list of integers. Either use the default buffer or set `*rbuf` to point to a larger buffer allocated with *driver_alloc*. The buffer will be freed automatically after `control` has returned.

Using binaries is faster if more than a few bytes are returned.

The return value is the number of bytes returned in `*rbuf`.

void (*timeout)(ErlDrvData drv_data)

This function is called any time after the driver's timer reaches 0. The timer is activated with `driver_set_timer`. There are no priorities or ordering among drivers, so if several drivers time out at the same time, any one of them is called first.

void (*outputv)(ErlDrvData drv_data, ErlIOVec *ev)

This function is called whenever the port is written to. If it is `NULL`, the `output` function is called instead. This function is faster than `output`, because it takes an `ErlIOVec` directly, which requires no copying of the data. The port should be in binary mode, see `open_port/2`.

The `ErlIOVec` contains both a `SysIOVec`, suitable for `writev`, and one or more binaries. If these binaries should be retained, when the driver returns from `outputv`, they can be queued (using *driver_enq_bin* for instance), or if they are kept in a static or global variable, the reference counter can be incremented.

void (*ready_async)(ErlDrvData drv_data, ErlDrvThreadData thread_data)

This function is called after an asynchronous call has completed. The asynchronous call is started with *driver_async*. This function is called from the erlang emulator thread, as opposed to the asynchronous function, which is called in some thread (if multithreading is enabled).

int (*call)(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen, unsigned int *flags)

This function is called from `erlang:port_call/3`. It works a lot like the `control` call-back, but uses the external term format for input and output.

`command` is an integer, obtained from the call from erlang (the second argument to `erlang:port_call/3`).

`buf` and `len` provide the arguments to the call (the third argument to `erlang:port_call/3`). They can be decoded using `ei` functions.

`rbuf` points to a return buffer, `rlen` bytes long. The return data should be a valid erlang term in the external (binary) format. This is converted to an erlang term and returned by `erlang:port_call/3` to the caller. If more space than `rlen` bytes is needed to return data, `*rbuf` can be set to memory allocated with `driver_alloc`. This memory will be freed automatically after `call` has returned.

The return value is the number of bytes returned in `*rbuf`. If `ERL_DRV_ERROR_GENERAL` is returned (or in fact, anything < 0), `erlang:port_call/3` will throw a `BAD_ARG`.

void (*event)(ErlDrvData drv_data, ErlDrvEvent event, ErlDrvEventData event_data)

Intentionally left undocumented.

int extended_marker

This field should either be equal to `ERL_DRV_EXTENDED_MARKER` or `0`. An old driver (not aware of the extended driver interface) should set this field to `0`. If this field is equal to `0`, all the fields following this field also *have* to be `0`, or `NULL` in case it is a pointer field.

int major_version

This field should equal `ERL_DRV_EXTENDED_MAJOR_VERSION` if the `extended_marker` field equals `ERL_DRV_EXTENDED_MARKER`.

int minor_version

This field should equal `ERL_DRV_EXTENDED_MINOR_VERSION` if the `extended_marker` field equals `ERL_DRV_EXTENDED_MARKER`.

int driver_flags

This field is used to pass driver capability information to the runtime system. If the `extended_marker` field equals `ERL_DRV_EXTENDED_MARKER`, it should contain `0` or driver flags (`ERL_DRV_FLAG_*`) ored bitwise. Currently the following driver flags exist:

`ERL_DRV_FLAG_USE_PORT_LOCKING`
The runtime system will use port level locking on all ports executing this driver instead of driver level locking when the driver is run in a runtime system with SMP support. For more information see the *erl_driver* documentation.

`ERL_DRV_FLAG_SOFT_BUSY`
Marks that driver instances can handle being called in the *output* and/or *outputv* callbacks even though a driver instance has marked itself as busy (see *set_busy_port()*). Since erts version 5.7.4 this flag is required for drivers used by the Erlang distribution (the behaviour has always been required by drivers used by the distribution).

void *handle2

This field is reserved for the emulators internal use. The emulator will modify this field; therefore, it is important that the `driver_entry` isn't declared `const`.

void (*process_exit)(ErlDrvData drv_data, ErlDrvMonitor *monitor)

This callback is called when a monitored process exits. The `drv_data` is the data associated with the port for which the process is monitored (using *driver_monitor_process*) and the `monitor` corresponds to the `ErlDrvMonitor` structure filled in when creating the monitor. The driver interface function *driver_get_monitored_process* can be used to retrieve the process id of the exiting process as an `ErlDrvTermData`.

void (*stop_select)(ErlDrvEvent event, void* reserved)

This function is called on behalf of *driver_select* when it is safe to close an event object.

A typical implementation on Unix is to do `close((int)event)`.

Argument `reserved` is intended for future use and should be ignored.

In contrast to most of the other call-back functions, `stop_select` is called independent of any port. No `ErlDrvData` argument is passed to the function. No driver lock or port lock is guaranteed to be held. The port that called `driver_select` might even be closed at the time `stop_select` is called. But it could also be the case that `stop_select` is called directly by `driver_select`.

It is not allowed to call any functions in the *driver API* from `stop_select`. This strict limitation is due to the volatile context that `stop_select` may be called.

SEE ALSO

erl_driver(3), *erl_ddll(3)*, *erlang(3)*, kernel(3)

erts_alloc

C Library

`erts_alloc` is an Erlang Run-Time System internal memory allocator library. `erts_alloc` provides the Erlang Run-Time System with a number of memory allocators.

Allocators

Currently the following allocators are present:

`temp_alloc`
: Allocator used for temporary allocations.

`eheap_alloc`
: Allocator used for Erlang heap data, such as Erlang process heaps.

`binary_alloc`
: Allocator used for Erlang binary data.

`ets_alloc`
: Allocator used for ETS data.

`driver_alloc`
: Allocator used for driver data.

`sl_alloc`
: Allocator used for memory blocks that are expected to be short-lived.

`ll_alloc`
: Allocator used for memory blocks that are expected to be long-lived, for example Erlang code.

`fix_alloc`
: A very fast allocator used for some fix-sized data. `fix_alloc` manages a set of memory pools from which memory blocks are handed out. `fix_alloc` allocates memory pools from `ll_alloc`. Memory pools that have been allocated are never deallocated.

`std_alloc`
: Allocator used for most memory blocks not allocated via any of the other allocators described above.

`sys_alloc`
: This is normally the default `malloc` implementation used on the specific OS.

`mseg_alloc`
: A memory segment allocator. `mseg_alloc` is used by other allocators for allocating memory segments and is currently only available on systems that have the `mmap` system call. Memory segments that are deallocated are kept for a while in a segment cache before they are destroyed. When segments are allocated, cached segments are used if possible instead of creating new segments. This in order to reduce the number of system calls made.

`sys_alloc` and `fix_alloc` are always enabled and cannot be disabled. `mseg_alloc` is always enabled if it is available and an allocator that uses it is enabled. All other allocators can be *enabled or disabled*. By default all allocators are enabled. When an allocator is disabled, `sys_alloc` is used instead of the disabled allocator.

The main idea with the `erts_alloc` library is to separate memory blocks that are used differently into different memory areas, and by this achieving less memory fragmentation. By putting less effort in finding a good fit for memory blocks that are frequently allocated than for those less frequently allocated, a performance gain can be achieved.

The alloc_util framework

Internally a framework called `alloc_util` is used for implementing allocators. `sys_alloc`, `fix_alloc`, and `mseg_alloc` do not use this framework; hence, the following does *not* apply to them.

An allocator manages multiple areas, called carriers, in which memory blocks are placed. A carrier is either placed in a separate memory segment (allocated via `mseg_alloc`) or in the heap segment (allocated via `sys_alloc`).

Multiblock carriers are used for storage of several blocks. Singleblock carriers are used for storage of one block. Blocks that are larger than the value of the singleblock carrier threshold (*sbct*) parameter are placed in singleblock carriers. Blocks smaller than the value of the `sbct` parameter are placed in multiblock carriers. Normally an allocator creates a "main multiblock carrier". Main multiblock carriers are never deallocated. The size of the main multiblock carrier is determined by the value of the *mmbcs* parameter.

Sizes of multiblock carriers allocated via `mseg_alloc` are decided based on the values of the largest multiblock carrier size (*lmbcs*), the smallest multiblock carrier size (*smbcs*), and the multiblock carrier growth stages (*mbcgs*) parameters. If `nc` is the current number of multiblock carriers (the main multiblock carrier excluded) managed by an allocator, the size of the next `mseg_alloc` multiblock carrier allocated by this allocator will roughly be `smbcs +nc*(lmbcs-smbcs)/mbcgs` when `nc <= mbcgs`, and `lmbcs` when `nc > mbcgs`. If the value of the `sbct` parameter should be larger than the value of the `lmbcs` parameter, the allocator may have to create multiblock carriers that are larger than the value of the `lmbcs` parameter, though. Singleblock carriers allocated via `mseg_alloc` are sized to whole pages.

Sizes of carriers allocated via `sys_alloc` are decided based on the value of the `sys_alloc` carrier size (*ycs*) parameter. The size of a carrier is the least number of multiples of the value of the `ycs` parameter that satisfies the request.

Coalescing of free blocks are always performed immediately. Boundary tags (headers and footers) in free blocks are used which makes the time complexity for coalescing constant.

The memory allocation strategy used for multiblock carriers by an allocator is configurable via the *as* parameter. Currently the following strategies are available:

Best fit

Strategy: Find the smallest block that satisfies the requested block size.

Implementation: A balanced binary search tree is used. The time complexity is proportional to log N, where N is the number of sizes of free blocks.

Address order best fit

Strategy: Find the smallest block that satisfies the requested block size. If multiple blocks are found, choose the one with the lowest address.

Implementation: A balanced binary search tree is used. The time complexity is proportional to log N, where N is the number of free blocks.

Good fit

Strategy: Try to find the best fit, but settle for the best fit found during a limited search.

Implementation: The implementation uses segregated free lists with a maximum block search depth (in each list) in order to find a good fit fast. When the maximum block search depth is small (by default 3) this implementation has a time complexity that is constant. The maximum block search depth is configurable via the *mbsd* parameter.

A fit

Strategy: Do not search for a fit, inspect only one free block to see if it satisfies the request. This strategy is only intended to be used for temporary allocations.

Implementation: Inspect the first block in a free-list. If it satisfies the request, it is used; otherwise, a new carrier is created. The implementation has a time complexity that is constant.

As of erts version 5.6.1 the emulator will refuse to use this strategy on other allocators than `temp_alloc`. This since it will only cause problems for other allocators.

System Flags Effecting erts_alloc

> ### Warning:
>
> Only use these flags if you are absolutely sure what you are doing. Unsuitable settings may cause serious performance degradation and even a system crash at any time during operation.

Memory allocator system flags have the following syntax: `+M<S><P> <V>` where `<S>` is a letter identifying a subsystem, `<P>` is a parameter, and `<V>` is the value to use. The flags can be passed to the Erlang emulator (*erl*) as command line arguments.

System flags effecting specific allocators have an upper-case letter as `<S>`. The following letters are used for the currently present allocators:

- `B: binary_alloc`
- `D: std_alloc`
- `E: ets_alloc`
- `F: fix_alloc`
- `H: eheap_alloc`
- `L: ll_alloc`
- `M: mseg_alloc`
- `R: driver_alloc`
- `S: sl_alloc`
- `T: temp_alloc`
- `Y: sys_alloc`

The following flags are available for configuration of `mseg_alloc`:

`+MMamcbf <size>`
: Absolute max cache bad fit (in kilobytes). A segment in the memory segment cache is not reused if its size exceeds the requested size with more than the value of this parameter. Default value is 4096.

`+MMrmcbf <ratio>`
: Relative max cache bad fit (in percent). A segment in the memory segment cache is not reused if its size exceeds the requested size with more than relative max cache bad fit percent of the requested size. Default value is 20.

`+MMmcs <amount>`
: Max cached segments. The maximum number of memory segments stored in the memory segment cache. Valid range is 0-30. Default value is 5.

`+MMcci <time>`
: Cache check interval (in milliseconds). The memory segment cache is checked for segments to destroy at an interval determined by this parameter. Default value is 1000.

The following flags are available for configuration of `fix_alloc`:

`+MFe true`
: Enable `fix_alloc`. Note: `fix_alloc` cannot be disabled.

The following flags are available for configuration of `sys_alloc`:

`+MYe true`
: Enable `sys_alloc`. Note: `sys_alloc` cannot be disabled.

`+MYm libc`
: `malloc` library to use. Currently only `libc` is available. `libc` enables the standard `libc` malloc implementation. By default `libc` is used.

`+MYtt <size>`
: Trim threshold size (in kilobytes). This is the maximum amount of free memory at the top of the heap (allocated by `sbrk`) that will be kept by `malloc` (not released to the operating system). When the amount of free memory at the top of the heap exceeds the trim threshold, `malloc` will release it (by calling `sbrk`). Trim threshold is given in kilobytes. Default trim threshold is 128. *Note:* This flag will only have any effect when the emulator has been linked with the GNU C library, and uses its `malloc` implementation.

`+MYtp <size>`
: Top pad size (in kilobytes). This is the amount of extra memory that will be allocated by `malloc` when `sbrk` is called to get more memory from the operating system. Default top pad size is 0. *Note:* This flag will only have any effect when the emulator has been linked with the GNU C library, and uses its `malloc` implementation.

The following flags are available for configuration of allocators based on `alloc_util`. If `u` is used as subsystem identifier (i.e., `<S> = u`) all allocators based on `alloc_util` will be effected. If `B`, `D`, `E`, `H`, `L`, `R`, `S`, or `T` is used as subsystem identifier, only the specific allocator identified will be effected:

`+M<S>as bf|aobf|gf|af`
: Allocation strategy. Valid strategies are `bf` (best fit), `aobf` (address order best fit), `gf` (good fit), and `af` (a fit). See *the description of allocation strategies* in "the `alloc_util` framework" section.

`+M<S>asbcst <size>`
: Absolute singleblock carrier shrink threshold (in kilobytes). When a block located in an `mseg_alloc` singleblock carrier is shrunk, the carrier will be left unchanged if the amount of unused memory is less than this threshold; otherwise, the carrier will be shrunk. See also *rsbcst*.

`+M<S>e true|false`
: Enable allocator `<S>`.

`+M<S>lmbcs <size>`
: Largest (`mseg_alloc`) multiblock carrier size (in kilobytes). See *the description on how sizes for mseg_alloc multiblock carriers are decided* in "the `alloc_util` framework" section.

`+M<S>mbcgs <ratio>`
: (`mseg_alloc`) multiblock carrier growth stages. See *the description on how sizes for mseg_alloc multiblock carriers are decided* in "the `alloc_util` framework" section.

`+M<S>mbsd <depth>`
: Max block search depth. This flag has effect only if the good fit strategy has been selected for allocator `<S>`. When the good fit strategy is used, free blocks are placed in segregated free-lists. Each free list contains blocks of sizes in a specific range. The max block search depth sets a limit on the maximum number of blocks to inspect in a free list during a search for suitable block satisfying the request.

`+M<S>mmbcs <size>`
: Main multiblock carrier size. Sets the size of the main multiblock carrier for allocator `<S>`. The main multiblock carrier is allocated via `sys_alloc` and is never deallocated.

`+M<S>mmmbc <amount>`
: Max `mseg_alloc` multiblock carriers. Maximum number of multiblock carriers allocated via `mseg_alloc` by allocator `<S>`. When this limit has been reached, new multiblock carriers will be allocated via `sys_alloc`.

`+M<S>mmsbc <amount>`
: Max `mseg_alloc` singleblock carriers. Maximum number of singleblock carriers allocated via `mseg_alloc` by allocator `<S>`. When this limit has been reached, new singleblock carriers will be allocated via `sys_alloc`.

`+M<S>ramv <bool>`
: Realloc always moves. When enabled, reallocate operations will more or less be translated into an allocate, copy, free sequence. This often reduce memory fragmentation, but costs performance.

`+M<S>rmbcmt <ratio>`
: Relative multiblock carrier move threshold (in percent). When a block located in a multiblock carrier is shrunk, the block will be moved if the ratio of the size of the returned memory compared to the previous size is more than this threshold; otherwise, the block will be shrunk at current location.

`+M<S>rsbcmt <ratio>`
: Relative singleblock carrier move threshold (in percent). When a block located in a singleblock carrier is shrunk to a size smaller than the value of the *sbct* parameter, the block will be left unchanged in the singleblock carrier if the ratio of unused memory is less than this threshold; otherwise, it will be moved into a multiblock carrier.

`+M<S>rsbcst <ratio>`
: Relative singleblock carrier shrink threshold (in percent). When a block located in an `mseg_alloc` singleblock carrier is shrunk, the carrier will be left unchanged if the ratio of unused memory is less than this threshold; otherwise, the carrier will be shrunk. See also *asbcst*.

`+M<S>sbct <size>`
: Singleblock carrier threshold. Blocks larger than this threshold will be placed in singleblock carriers. Blocks smaller than this threshold will be placed in multiblock carriers.

`+M<S>smbcs <size>`
: Smallest (`mseg_alloc`) multiblock carrier size (in kilobytes). See *the description on how sizes for mseg_alloc multiblock carriers are decided* in "the `alloc_util` framework" section.

`+M<S>t true|false|<amount>`
: Multiple, thread specific instances of the allocator. This option will only have any effect on the runtime system with SMP support. Default behaviour on the runtime system with SMP support (`N` equals the number of scheduler threads):

  `temp_alloc`
  `N + 1` instances.
  `ll_alloc`
  `1` instance.
  Other allocators
  `N` instances when `N` is less than or equal to `16`. `16` instances when `N` is greater than `16`.

  `temp_alloc` will always use `N + 1` instances when this option has been enabled regardless of the amount passed. Other allocators will use the same amount of instances as the amount passed as long as it isn't greater than `N`.

Currently the following flags are available for configuration of `alloc_util`, i.e. all allocators based on `alloc_util` will be effected:

`+Muycs <size>`
: `sys_alloc` carrier size. Carriers allocated via `sys_alloc` will be allocated in sizes which are multiples of the `sys_alloc` carrier size. This is not true for main multiblock carriers and carriers allocated during a memory shortage, though.

`+Mummc <amount>`
: Max `mseg_alloc` carriers. Maximum number of carriers placed in separate memory segments. When this limit has been reached, new carriers will be placed in memory retrieved from `sys_alloc`.

Instrumentation flags:

`+Mim true|false`
: A map over current allocations is kept by the emulator. The allocation map can be retrieved via the `instrument` module. `+Mim true` implies `+Mis true`. `+Mim true` is the same as *-instr*.

`+Mis true|false`
: Status over allocated memory is kept by the emulator. The allocation status can be retrieved via the `instrument` module.

`+Mit X`
: Reserved for future use. Do *not* use this flag.

> ### Note:
>
> When instrumentation of the emulator is enabled, the emulator uses more memory and runs slower.

Other flags:

`+Mea min|max|r9c|r10b|r11b|config`

`min`
Disables all allocators that can be disabled.

`max`
Enables all allocators (currently default).

`r9c|r10b|r11b`
Configures all allocators as they were configured in respective OTP release. These will eventually be removed.

`config`
Disables features that cannot be enabled while creating an allocator configuration with *erts_alloc_config(3)*. Note, this option should only be used while running `erts_alloc_config`, *not* when using the created configuration.

Only some default values have been presented here. *erlang:system_info(allocator)*, and *erlang:system_info({allocator, Alloc})* can be used in order to obtain currently used settings and current status of the allocators.

> ### Note:
>
> Most of these flags are highly implementation dependent, and they may be changed or removed without prior notice.
>
> `erts_alloc` is not obliged to strictly use the settings that have been passed to it (it may even ignore them).

erts_alloc_config(3) is a tool that can be used to aid creation of an `erts_alloc` configuration that is suitable for a limited number of runtime scenarios.

SEE ALSO

erts_alloc_config(3), *erl(1)*, *instrument(3)*, *erlang(3)*

erl_nif

C Library

> ### Warning:
>
> The NIF concept was introduced in R13B03 as an EXPERIMENTAL feature. The interfaces may be changed in any way in coming releases. The plan is however to lift the experimental label and maintain interface backward compatibility from R14B.
>
> Incompatible changes in *R13B04*:
>
> - The function prototypes of the NIFs have changed to expect `argc` and `argv` arguments. The arity of a NIF is by that no longer limited to 3.
> - `enif_get_data` renamed as `enif_priv_data`.
> - `enif_make_string` got a third argument for character encoding.

A NIF library contains native implementation of some functions of an Erlang module. The native implemented functions (NIFs) are called like any other functions without any difference to the caller. Each NIF must also have an implementation in Erlang that will be invoked if the function is called before the NIF library has been successfully loaded. A typical such stub implementation is to throw an exception. But it can also be used as a fallback implementation if the NIF library is not implemented for some architecture.

A minimal example of a NIF library can look like this:

```
/* niftest.c */
#include "erl_nif.h"

static ERL_NIF_TERM hello(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    return enif_make_string(env, "Hello world!", ERL_NIF_LATIN1);
}

static ErlNifFunc nif_funcs[] =
{
    {"hello", 0, hello}
};

ERL_NIF_INIT(niftest,nif_funcs,NULL,NULL,NULL,NULL)
```

and the Erlang module would have to look something like this:

```
-module(niftest).

-export([init/0, hello/0]).

init() ->
      erlang:load_nif("./niftest", 0).

hello() ->
      "NIF library not loaded".
```

and compile and test something like this (on Linux):

```
$> gcc -fPIC -shared -o niftest.so niftest.c -I $ERL_ROOT/usr/include/
$> erl

1> c(niftest).
{ok,niftest}
2> niftest:hello().
"NIF library not loaded"
3> niftest:init().
ok
4> niftest:hello().
"Hello world!"
```

A better solution for a real module is to take advantage of the new directive *on_load* to automatically load the NIF library when the module is loaded.

> Note:
>
> A NIF must be exported or used locally by the module (or both). An unused local stub function will be optimized away by the compiler causing loading of the NIF library to fail.

A loaded NIF library is tied to the Erlang module code version that loaded it. If the module is upgraded with a new version, the new Erlang code will have to load its own NIF library (or maybe choose not to). The new code version can however choose to load the exact same NIF library as the old code if it wants to. Sharing the same dynamic library will mean that static data defined by the library will be shared as well. To avoid unintentionally shared static data, each Erlang module code can keep its own private data. This private data can be set when the NIF library is loaded and then retrieved by calling *enif_priv_data()*.

There is no way to explicitly unload a NIF library. A library will be automatically unloaded when the module code that it belongs to is purged by the code server. A NIF library will also be unloaded if it is replaced by another version of the library by a second call to `erlang:load_nif/2` from the same module code.

FUNCTIONALITY

All functions that a NIF library needs to do with Erlang are performed through the NIF API functions. There are functions for the following functionality:

Read and write Erlang terms

> Any Erlang terms can be passed to a NIF as function arguments and be returned as function return values. The terms are of C-type `ERL_NIF_TERM` and can only be read or written using API functions. Most functions to read the content of a term are prefixed `enif_get_` and usually return true (or false) if the term was of the expected type (or not). The functions to write terms are all prefixed `enif_make_` and usually return the created `ERL_NIF_TERM`. There are also some functions to query terms, like `enif_is_atom`, `enif_is_identical` and `enif_compare`.

Binaries

> Terms of type binary are accessed with the help of the struct type *ErlNifBinary* that contains a pointer (`data`) to the raw binary data and the length (`size`) of the data in bytes. Both `data` and `size` are read-only and should only be written using calls to API functions. Instances of `ErlNifBinary` are however always allocated by the user (usually as local variables).

The raw data pointed to by `data` is only mutable after a call to *enif_alloc_binary* or *enif_realloc_binary*. All other functions that operates on a binary will leave the data as read-only. A mutable binary must in the end either be freed with *enif_release_binary* or made read-only by transferring it to an Erlang term with *enif_make_binary*. But it does not have do happen in the same NIF call. Read-only binaries does not have to be released.

Binaries are sequences of whole bytes. Bitstrings with an arbitrary bit length have no support yet.

Resource objects

The use of resource objects is a way to return pointers to native data structures from a NIF in a safe way. A resource object is just a block of memory allocated with *enif_alloc_resource()*. A handle ("safe pointer") to this memory block can then be returned to Erlang by the use of *enif_make_resource()*. The term returned by `enif_make_resource` is totally opaque in nature. It can be stored and passed between processses on the same node, but the only real end usage is to pass it back as argument to a NIF. The NIF can then do *enif_get_resource()* and get back a pointer to the memory block that is guaranteed to still be valid. A resource object will not be deallocated until the last handle term has been garbage collected by the VM and the resource has been released with *enif_release_resource()* (not necessarily in that order).

All resource objects are created as instances of some *resource type*. This makes resources from different modules to be distinguishable. A resource type is created by calling *enif_open_resource_type()* when a library is loaded. Objects of that resource type can then later be allocated and `enif_get_resource` verifies that the resource is of the expected type. A resource type can have a user supplied destructor function that is automatically called when resources of that type are released (by either the garbage collector or `enif_release_resource`). Resource types are uniquely identified by a supplied name string.

Resource types support upgrade in runtime by allowing a loaded NIF library to takeover an already existing resource type and thereby "inherit" all existing objects of that type. The destructor of the new library will thereafter be called for the inherited objects and the library with the old destructor function can be safely unloaded. Existing resource objects, of a module that is upgraded, must either be deleted or taken over by the new NIF library. The unloading of a library will be postponed as long as it exists resource objects with a destructor function in the library.

Here is a template example of how to create and return a resource object.

```
    ERL_NIF_TERM term;
    MyStruct* ptr = enif_alloc_resource(env, my_resource_type, sizeof(MyStruct));

    /* initialize struct ... */

    term = enif_make_resource(env, ptr);

    if (keep_a_reference_of_our_own) {
        /* store 'ptr' in static variable, private data or other resource object */
    }
    else {
        enif_release_resource(env, obj);
        /* resource now only owned by "Erlang" */
    }
    return term;
}
```

Threads and concurrency

A NIF is thread-safe without any explicit synchronization as long as it acts as a pure function and only reads the supplied arguments. As soon as you write towards a shared state either through static variables or *enif_priv_data* you need to supply your own explicit synchronization. Resource objects will also require synchronization if you treat them as mutable.

The library initialization callbacks `load`, `reload` and `upgrade` are all thread-safe even for shared state data.

Avoid doing lengthy work in NIF calls as that may degrade the responsiveness of the VM. NIFs are called directly by the same scheduler thread that executed the calling Erlang code. The calling scheduler will thus be blocked from doing any other work until the NIF returns.

INITIALIZATION

ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)

This is the magic macro to initialize a NIF library. It should be evaluated in global file scope.

`MODULE` is the name of the Erlang module as an identifier without string quotations. It will be stringified by the macro.

`funcs` is a static array of function descriptors for all the implemented NIFs in this library.

`load`, `reload`, `upgrade` and `unload` are pointers to functions. One of `load`, `reload` or `upgrade` will be called to initialize the library. `unload` is called to release the library. They are all described individually below.

int (*load)(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info)

`load` is called when the NIF library is loaded and there is no previously loaded library for this module.

`*priv_data` can be set to point to some private data that the library needs in able to keep a state between NIF calls. `enif_priv_data()` will return this pointer. `*priv_data` will be initialized to NULL when `load` is called.

`load_info` is the second argument to *erlang:load_nif/2*.

The library will fail to load if `load` returns anything other than 0. `load` can be NULL in case no initialization is needed.

int (*reload)(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info)

`reload` is called when the NIF library is loaded and there is already a previously loaded library for this module code.

Works the same as `load`. The only difference is that `*priv_data` already contains the value set by the previous call to `load` or `reload`.

The library will fail to load if `reload` returns anything other than 0 or if `reload` is NULL.

int (*upgrade)(ErlNifEnv* env, void** priv_data, void** old_priv_data, ERL_NIF_TERM load_info)

`upgrade` is called when the NIF library is loaded and there is no previously loaded library for this module code, BUT there is old code of this module with a loaded NIF library.

Works the same as `load`. The only difference is that `*old_priv_data` already contains the value set by the last call to `load` or `reload` for the old module code. `*priv_data` will be initialized to NULL when `upgrade` is called. It is allowed to write to both *priv_data and *old_priv_data.

The library will fail to load if `upgrade` returns anything other than 0 or if `upgrade` is NULL.

void (*unload)(ErlNifEnv* env, void* priv_data)

`unload` is called when the module code that the NIF library belongs to is purged as old. New code of the same module may or may not exist. Note that `unload` is not called for a replaced library as a consequence of `reload`.

DATA TYPES

ERL_NIF_TERM

Variables of type `ERL_NIF_TERM` can refer to any Erlang term. This is an opaque type and values of it can only by used either as arguments to API functions or as return values from NIFs. A variable of type `ERL_NIF_TERM` is only valid until the NIF call, where it was obtained, returns.

ErlNifEnv

`ErlNifEnv` contains information about the context in which a NIF call is made. This pointer should not be dereferenced in any way, but only passed on to API functions. An `ErlNifEnv` pointer is only valid until the function, where is what supplied as argument, returns. There is thus useless and dangerous to store `ErlNifEnv` pointers in between NIF calls.

ErlNifFunc

```
typedef struct {
    const char* ;
    unsigned ;
    ERL_NIF_TERM (*)(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]);
} ErlNifFunc;
```

Describes a NIF by its name, arity and implementation. `fptr` is a pointer to the function that implements the NIF. The argument `argv` of a NIF will contain the function arguments passed to the NIF and `argc` is the length of the array, i.e. the function arity. `argv[N-1]` will thus denote the Nth argument to the NIF. Note that the `argc` argument allows for the same C function to implement several Erlang functions with different arity (but same name probably).

ErlNifBinary

```
typedef struct {
    unsigned ;
    unsigned char* ;
} ErlNifBinary;
```

`ErlNifBinary` contains transient information about an inspected binary term. `data` is a pointer to a buffer of `size` bytes with the raw content of the binary.

ErlNifResourceType

Each instance of `ErlNifResourceType` represent a class of memory managed resource objects that can be garbage collected. Each resource type has a unique name and a destructor function that is called when objects of its type are released.

ErlNifResourceDtor

```
typedef void ErlNifResourceDtor(ErlNifEnv* env, void* obj);
```

The function prototype of a resource destructor function. A destructor function is not allowed to call any term-making functions.

ErlNifCharEncoding

```
typedef enum {
    ERL_NIF_LATIN1
}ErlNifCharEncoding;
```

The character encoding used in strings. The only supported encoding is currently `ERL_NIF_LATIN1` for iso-latin-1 (8-bit ascii).

ErlNifSysInfo

Used by *enif_system_info* to return information about the runtime system. Contains currently the exact same content as *ErlDrvSysInfo*.

Exports

`void* enif_alloc(ErlNifEnv* env, size_t size)`

Allocate memory of `size` bytes. Return NULL if allocation failed.

`int enif_alloc_binary(ErlNifEnv* env, unsigned size, ErlNifBinary* bin)`

Allocate a new binary of size of `size` bytes. Initialize the structure pointed to by `bin` to refer to the allocated binary. The binary must either be released by *enif_release_binary()* or ownership transferred to an Erlang term with *enif_make_binary()*. An allocated (and owned) `ErlNifBinary` can be kept between NIF calls.

Return false if allocation failed.

`void* enif_alloc_resource(ErlNifEnv* env, ErlNifResourceType* type, unsigned size)`

Allocate a memory managed resource object of type `type` and size `size` bytes.

`int enif_compare(ErlNifEnv* env, ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)`

Return an integer less than, equal to, or greater than zero if `lhs` is found, respectively, to be less than, equal, or greater than `rhs`. Corresponds to the Erlang operators ==, `/=`, =<, <, >= and > (but *not* `=:=` or `=/=`).

`void enif_cond_broadcast(ErlNifCond *cnd)`

Same as *erl_drv_cond_broadcast()*.

`ErlNifCond* enif_cond_create(char *name)`

Same as *erl_drv_cond_create()*.

`void enif_cond_destroy(ErlNifCond *cnd)`

Same as *erl_drv_cond_destroy()*.

`void enif_cond_signal(ErlNifCond *cnd)`

Same as *erl_drv_cond_signal()*.

`void enif_cond_wait(ErlNifCond *cnd, ErlNifMutex *mtx)`

Same as *erl_drv_cond_wait()*.

int enif_equal_tids(ErlNifTid tid1, ErlNifTid tid2)

Same as *erl_drv_equal_tids()*.

void enif_free(ErlNifEnv* env, void* ptr)

Free memory allocated by `enif_alloc`.

int enif_get_atom(ErlNifEnv* env, ERL_NIF_TERM term, char* buf, unsigned size)

Write a null-terminated string, in the buffer pointed to by `buf` of size `size`, consisting of the string representation of the atom `term`. Return the number of bytes written (including terminating null character) or 0 if `term` is not an atom with maximum length of `size-1`.

int enif_get_double(ErlNifEnv* env, ERL_NIF_TERM term, double* dp)

Set `*dp` to the floating point value of `term` or return false if `term` is not a float.

int enif_get_int(ErlNifEnv* env, ERL_NIF_TERM term, int* ip)

Set `*ip` to the integer value of `term` or return false if `term` is not an integer or is outside the bounds of type `int`

int enif_get_list_cell(ErlNifEnv* env, ERL_NIF_TERM list, ERL_NIF_TERM* head, ERL_NIF_TERM* tail)

Set `*head` and `*tail` from `list` or return false if `list` is not a non-empty list.

int enif_get_long(ErlNifEnv* env, ERL_NIF_TERM term, long int* ip)

Set `*ip` to the long integer value of `term` or return false if `term` is not an integer or is outside the bounds of type `long int`.

int enif_get_resource(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifResourceType* type, void** objp)

Set `*objp` to point to the resource object referred to by `term`. The pointer is valid until the calling NIF returns and should not be released.

Return false if `term` is not a handle to a resource object of type `type`.

int enif_get_string(ErlNifEnv* env, ERL_NIF_TERM list, char* buf, unsigned size, ErlNifCharEncoding encode)

Write a null-terminated string, in the buffer pointed to by `buf` with size `size`, consisting of the characters in the string `list`. The characters are written using encoding *encode*. Return the number of bytes written (including terminating null character), or `-size` if the string was truncated due to buffer space, or 0 if `list` is not a string that can be encoded with `encode` or if `size` was less than 1. The written string is always null-terminated unless buffer `size` is less than 1.

int enif_get_tuple(ErlNifEnv* env, ERL_NIF_TERM term, int* arity, const ERL_NIF_TERM** array)

If `term` is a tuple, set `*array` to point to an array containing the elements of the tuple and set `*arity` to the number of elements. Note that the array is read-only and `(*array)[N-1]` will be the Nth element of the tuple. `*array` is undefined if the arity of the tuple is zero.

Return false if `term` is not a tuple.

`int enif_get_uint(ErlNifEnv* env, ERL_NIF_TERM term, unsigned int* ip)`

Set `*ip` to the unsigned integer value of `term` or return false if `term` is not an unsigned integer or is outside the bounds of type `unsigned int`

`int enif_get_ulong(ErlNifEnv* env, ERL_NIF_TERM term, unsigned long* ip)`

Set `*ip` to the unsigned long integer value of `term` or return false if `term` is not an unsigned integer or is outside the bounds of type `unsigned long`

`int enif_inspect_binary(ErlNifEnv* env, ERL_NIF_TERM bin_term, ErlNifBinary* bin)`

Initialize the structure pointed to by `bin` with information about the binary term `bin_term`. Return false if `bin_term` is not a binary.

`int enif_inspect_iolist_as_binary(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifBinary* bin)`

Initialize the structure pointed to by `bin` with one continuous buffer with the same byte content as `iolist`. As with inspect_binary, the data pointed to by `bin` is transient and does not need to be released. Return false if `iolist` is not an iolist.

`int enif_is_atom(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is an atom.

`int enif_is_binary(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is a binary

`int enif_is_empty_list(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is an empty list.

`int enif_is_fun(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is a fun.

`int enif_is_identical(ErlNifEnv* env, ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)`

Return true if the two terms are identical. Corresponds to the Erlang operators `=:=` and `=/=`.

`int enif_is_pid(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is a pid.

`int enif_is_port(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is a port.

`int enif_is_ref(ErlNifEnv* env, ERL_NIF_TERM term)`

Return true if `term` is a reference.

ERL_NIF_TERM enif_make_atom(ErlNifEnv* env, const char* name)

Create an atom term from the C-string `name`. Unlike other terms, atom terms may be saved and used between NIF calls.

ERL_NIF_TERM enif_make_badarg(ErlNifEnv* env)

Make a badarg exception to be returned from a NIF.

ERL_NIF_TERM enif_make_binary(ErlNifEnv* env, ErlNifBinary* bin)

Make a binary term from `bin`. Any ownership of the binary data will be transferred to the created term and `bin` should be considered read-only for the rest of the NIF call and then as released.

ERL_NIF_TERM enif_make_double(ErlNifEnv* env, double d)

Create an floating-point term from a `double`.

int enif_make_existing_atom(ErlNifEnv* env, const char* name, ERL_NIF_TERM* atom)

Try to create the term of an already existing atom from the C-string `name`. If the atom already exist store the term in `*atom` and return true, otherwise return false.

ERL_NIF_TERM enif_make_int(ErlNifEnv* env, int i)

Create an integer term.

ERL_NIF_TERM enif_make_list(ErlNifEnv* env, unsigned cnt, ...)

Create an ordinary list term of length `cnt`. Expects `cnt` number of arguments (after `cnt`) of type ERL_NIF_TERM as the elements of the list. An empty list is returned if `cnt` is 0.

```
ERL_NIF_TERM enif_make_list1(ErlNifEnv* env, ERL_NIF_TERM e1)
ERL_NIF_TERM enif_make_list2(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM
e2)
ERL_NIF_TERM enif_make_list3(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM
e2, ERL_NIF_TERM e3)
ERL_NIF_TERM enif_make_list4(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e4)
ERL_NIF_TERM enif_make_list5(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e5)
ERL_NIF_TERM enif_make_list6(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e6)
ERL_NIF_TERM enif_make_list7(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e7)
ERL_NIF_TERM enif_make_list8(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e8)
ERL_NIF_TERM enif_make_list9(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e9)
```

Create an ordinary list term with length indicated by the function name. Prefer these functions (macros) over the variadic `enif_make_list` to get compile time error if the number of arguments does not match.

ERL_NIF_TERM enif_make_list_cell(ErlNifEnv* env, ERL_NIF_TERM head, ERL_NIF_TERM tail)

Create a list cell `[head | tail]`.

ERL_NIF_TERM enif_make_list_from_array(ErlNifEnv* env, const ERL_NIF_TERM arr[], unsigned cnt)

Create an ordinary list containing the elements of array `arr` of length `cnt`. An empty list is returned if `cnt` is 0.

ERL_NIF_TERM enif_make_long(ErlNifEnv* env, long int i)

Create an integer term from a `long int`.

ERL_NIF_TERM enif_make_ref(ErlNifEnv* env)

Create a reference like *erlang:make_ref/0*.

ERL_NIF_TERM enif_make_resource(ErlNifEnv* env, void* obj)

Create an opaque handle to a memory managed resource object obtained by *enif_alloc_resource*. No ownership transfer is done, the resource object still needs to be released by *enif_release_resource*.

Note that the only defined behaviour when using of a resource term in an Erlang program is to store it and send it between processes on the same node. Other operations such as matching or `term_to_binary` will have unpredictable (but harmless) results.

ERL_NIF_TERM enif_make_string(ErlNifEnv* env, const char* string, ErlNifCharEncoding encoding)

Create a list containing the characters of the null-terminated string `string` with encoding *encoding*.

ERL_NIF_TERM enif_make_sub_binary(ErlNifEnv* env, ERL_NIF_TERM bin_term, unsigned pos, unsigned size)

Make a subbinary of binary `bin_term`, starting at zero-based position `pos` with a length of `size` bytes. `bin_term` must be a binary or bitstring and `pos+size` must be less or equal to the number of whole bytes in `bin_term`.

ERL_NIF_TERM enif_make_tuple(ErlNifEnv* env, unsigned cnt, ...)

Create a tuple term of arity `cnt`. Expects `cnt` number of arguments (after `cnt`) of type ERL_NIF_TERM as the elements of the tuple.

```
ERL_NIF_TERM enif_make_tuple1(ErlNifEnv* env, ERL_NIF_TERM e1)
ERL_NIF_TERM enif_make_tuple2(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM
e2)
ERL_NIF_TERM enif_make_tuple3(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM
e2, ERL_NIF_TERM e3)
ERL_NIF_TERM enif_make_tuple4(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e4)
ERL_NIF_TERM enif_make_tuple5(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e5)
ERL_NIF_TERM enif_make_tuple6(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e6)
```

ERL_NIF_TERM enif_make_tuple7(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e7)
ERL_NIF_TERM enif_make_tuple8(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e8)
ERL_NIF_TERM enif_make_tuple9(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e9)

Create a tuple term with length indicated by the function name. Prefer these functions (macros) over the variadic `enif_make_tuple` to get compile time error if the number of arguments does not match.

ERL_NIF_TERM enif_make_tuple_from_array(ErlNifEnv* env, const ERL_NIF_TERM arr[], unsigned cnt)

Create a tuple containing the elements of array `arr` of length `cnt`.

ERL_NIF_TERM enif_make_uint(ErlNifEnv* env, unsigned int i)

Create an integer term from an `unsigned int`.

ERL_NIF_TERM enif_make_ulong(ErlNifEnv* env, unsigned long i)

Create an integer term from an `unsigned long int`.

ErlNifMutex* enif_mutex_create(char *name)

Same as *erl_drv_mutex_create()*.

void enif_mutex_destroy(ErlNifMutex *mtx)

Same as *erl_drv_mutex_destroy()*.

void enif_mutex_lock(ErlNifMutex *mtx)

Same as *erl_drv_mutex_lock()*.

int enif_mutex_trylock(ErlNifMutex *mtx)

Same as *erl_drv_mutex_trylock()*.

void enif_mutex_unlock(ErlNifMutex *mtx)

Same as *erl_drv_mutex_unlock()*.

ErlNifResourceType* enif_open_resource_type(ErlNifEnv* env, const char* name, ErlNifResourceDtor* dtor, ErlNifResourceFlags flags, ErlNifResourceFlags* tried)

Create or takeover a resource type identified by the string `name` and give it the destructor function pointed to by *dtor*. Argument `flags` can have the following values:

`ERL_NIF_RT_CREATE`
: Create a new resource type that does not already exist.

`ERL_NIF_RT_TAKEOVER`
: Open an existing resource type and take over ownership of all its instances. The supplied destructor `dtor` will be called both for existing instances as well as new instances not yet created by the calling NIF library.

The two flag values can be combined with bitwise-or. To avoid unintentionally name clashes a good practice is to include the module name as part of the type `name`. The `dtor` may be `NULL` in case no destructor is needed.

On success, return a pointer to the resource type and `*tried` will be set to either `ERL_NIF_RT_CREATE` or `ERL_NIF_RT_TAKEOVER` to indicate what was actually done. On failure, return `NULL` and set `*tried` to `flags`. It is allowed to set `tried` to `NULL`.

Note that `enif_open_resource_type` is only allowed to be called in the three callbacks *load*, *reload* and *upgrade*.

`void* enif_priv_data(ErlNifEnv* env)`

Return the pointer to the private data that was set by `load`, `reload` or `upgrade`.

Was previously named `enif_get_data`.

`void enif_realloc_binary(ErlNifEnv* env, ErlNifBinary* bin, unsigned size)`

Change the size of a binary `bin`. The source binary may be read-only, in which case it will be left untouched and a mutable copy is allocated and assigned to `*bin`.

`void enif_release_binary(ErlNifEnv* env, ErlNifBinary* bin)`

Release a binary obtained from `enif_alloc_binary`.

`void enif_release_resource(ErlNifEnv* env, void* obj)`

Release a resource objects obtained from `enif_alloc_resource`. The object may still be alive if it is referred to by Erlang terms. Each call to `enif_release_resource` must correspond to a previous call to `enif_alloc_resource`. References made by `enif_make_resource` can only be released by the garbage collector.

`ErlNifRWLock* enif_rwlock_create(char *name)`

Same as *erl_drv_rwlock_create()*.

`void enif_rwlock_destroy(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_destroy()*.

`void enif_rwlock_rlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_rlock()*.

`void enif_rwlock_runlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_runlock()*.

`void enif_rwlock_rwlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_rwlock()*.

`void enif_rwlock_rwunlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_rwunlock()*.

`int enif_rwlock_tryrlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_tryrlock()*.

`int enif_rwlock_tryrwlock(ErlNifRWLock *rwlck)`

Same as *erl_drv_rwlock_tryrwlock()*.

`unsigned enif_sizeof_resource(ErlNifEnv* env, void* obj)`

Get the byte size of a resource object `obj` obtained by `enif_alloc_resource`.

`void enif_system_info(ErlNifSysInfo *sys_info_ptr, size_t size)`

Same as *driver_system_info()*.

`int enif_thread_create(char *name,ErlNifTid *tid,void * (*func)(void *),void *args,ErlNifThreadOpts *opts)`

Same as *erl_drv_thread_create()*.

`void enif_thread_exit(void *resp)`

Same as *erl_drv_thread_exit()*.

`int enif_thread_join(ErlNifTid, void **respp)`

Same as *erl_drv_thread_join ()*.

`ErlNifThreadOpts* enif_thread_opts_create(char *name)`

Same as *erl_drv_thread_opts_create()*.

`void enif_thread_opts_destroy(ErlNifThreadOpts *opts)`

Same as *erl_drv_thread_opts_destroy()*.

`ErlNifTid enif_thread_self(void)`

Same as *erl_drv_thread_self()*.

`int enif_tsd_key_create(char *name, ErlNifTSDKey *key)`

Same as *erl_drv_tsd_key_create()*.

`void enif_tsd_key_destroy(ErlNifTSDKey key)`

Same as *erl_drv_tsd_key_destroy()*.

`void* enif_tsd_get(ErlNifTSDKey key)`

Same as *erl_drv_tsd_get()*.

`void enif_tsd_set(ErlNifTSDKey key, void *data)`

Same as *erl_drv_tsd_set()*.

SEE ALSO

load_nif(3)