Erlang Run-Time System Application (ERTS)

version 5.4

Contents

Erlang Run-Time System Application (ERTS)

Chapter 1

ERTS 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` | `process_dump` | `enable_trace` | `disable_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 for this function call, 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, 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 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.

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.

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, [P])`, where P 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). 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, [P])`, where P 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. Returns `true` and 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 selectiing 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 messsage 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 hugh 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) [page 70].

- "<*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) [page 70] 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 [page 12].

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 identitifiers) 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>,
(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 distibution. The distribution is normally carried by the TCP/IP protocol. Whats explained here is the method for replacing TCP/IP whith 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.

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 callback 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 callback routines execute in the main thread of the erlang machine, the callback functions can perform no blocking activity whatsoever. The callbacks should only set up file descriptors for waiting and/or read/write available data. All I/O has to be non blocking. Driver callbacks are however executed in sequence, why a global state can safely be updated within the routines.

When the driver is implemented, one would preferrably 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 `gen_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 throug a driver and an erlang interface module, a distribution module can be written. The distribution module is a module with well defined callbacks, much like a `gen_server` (there is no compiler support for checking the callbacks 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 bootscript 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 boottime (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 datatypes 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`, dont 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 avalable 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 callback has to have a special name and will infor the emulator of what callbacks 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 datatype large enough to hold a pointer. The pointer returned by this function will be the first argument to all other callbacks concerning this particular port. It is usually called the port handle. The emulator only stores the handle and doues 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 callback 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 syncronous way to drivers, described below. There can be an additional callback function for handling data that is frgmented (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 callback, so we wont.

- When a file descriptor is signaled for input. This callback 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 callback (when reading is actually possible). The typical scenario is that `driver_select` is called when an erlang process orderes 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 callback 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 readu for output, this callback is called an the driver can try to send the output. There may of course be queueing 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 callbacks call a `driver_failure_XXX` routine, this routine will be immediately called and the callback 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:driver_control/2`, which is a syncronous 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 callback 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 mainataining, variable length packet oriented protocol. All error correction, resending 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 dont need to code the integer in some special endianess, but i'll do it anyway brcause in most situation you do need to do it. Unix domain sockets are reliable and order maintaining, so we dont 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 descriptor
(31)                                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) };
```

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 erlangs driver queues to correspond to the operating systems dito, which is very convenient.

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

The driver structure is similar for statically linked in drivers an 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 callback for timers is needed. The last field (line 40) can be used to implement an interface similar to Unix `writev` for output. There is no need for such interface in a distribution driver, so we leave it with a NULL value (We will however use scatter/gather I/O internally in the driver).

Our defined callbacks 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 callback 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` callback, 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 callback 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 differnt 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, ther will be no data pumping done on this socket, but there will be read data avalable 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 ann accepting dito 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. Ther 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 (reguler) file for two purpouses:
  - 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 emulatorors 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 lockfile 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 lockfile + 1) rem 4. This creation value is also written back into the lockfile, 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 driver_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 callback 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 callbacks 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 asyncronous commands when the port is in *command mode* as well as the sending of all data when the port is in *data mode* is handeled in thi9s 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 veriable 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 idientification 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 soime kind of streamed mode, as this driver is for the distribuion 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` callback 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, whinc 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 que 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 buissiness of connectiong (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 syncronous interface called when erlang calls `erlang:driver_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 intermidiate 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 numer 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.
- '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)                              \n( 6)    do {

(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 lin 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 bu 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 dito.

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 shuld 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`}, denoting syntactically incorrect forms, 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 guard expressions:

- 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)}]`}.

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 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 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 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 Gs is a nonempty sequence of guard tests `G_1, ..., G_k`, and Rep(Gs) = `[Rep(G_1), ...,
Rep(G_k)]`.

A guard sequence Gss is a sequence of guards `Gs_1; ...; Gs_k`, and Rep(Gss) = `[Rep(Gs_1), ...,
Rep(Gs_k)]`. If the guard sequence is empty, Rep(Gss) = `[]`.

A guard test G is either `true`, an application of a BIF to a sequence of guard expressions (syntactically
this includes guard record tests), or a binary operator applied to two guard expressions.

- If G is `true`, then Rep(G) = {`atom,LINE,true`}.
- If G is an application `A(E_1, ..., E_k)`, where `A` is an atom and `E_1, ..., E_k` are guard
  expressions, then Rep(G) = {`call,LINE,{atom,LINE,A},[Rep(E_1), ..., Rep(E_k)]`}.
- If G is an operator expression `E_1 Op E_2`, where `Op` is a binary operator, and `E_1, E_2` are guard
  expressions, then Rep(G) = {`op,LINE,Op,Rep(E_1),Rep(E_2)`}.

All guard expressions are expressions and are represented in the same way as the corresponding expressions.

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

continued ...

... continued

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 1.2: 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

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 (DLL), 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

...

1.6.3 Compiling and linking a driver

...

1.6.4 Calling a driver as a port in erlang

...

1.6.5 The driver structure

The header file `erl_driver.h` contains all types, macros and prototypes needed for the driver.

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.)

The driver structure contains the name of the driver and some 15 function pointers. These pointers are called at different times by the emulator. Here is the declaration of `driver_entry`:

```
typedef struct erl_drv_entry {
    int (*init)(void);              /* called at system start up for statically
                                       linked drivers, and after loading for
                                       dynamically loaded drivers */
    ErlDrvData (*start)(ErlDrvPort port, char *command);
                                    /* called when open_port/2 is invoked.
                                       return value -1 means failure. */
    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;                  /* not used -- here for backwards compatibility */
    int (*control)(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);
                                    /* "ioctl" for drivers - invoked by
                                       port_command/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 thred_data);
} ErlDrvEntry;
```

1.6.6 Driver callbacks

The erlang emulator has callbacks that the driver uses.

1.6.7   Threads and drivers

...

1.6.8   Drivers on specific platforms

Solaris

...

Windows

...

VxWorks

...

1.6.9   Loading drivers

...

1.6.10   Preloaded drivers

...

1.6.11   Handling the binary term format with ei

Introduction

`ei` is a small set of C routines to encode and decode the erlang binary term format. It is just some small functions that encodes and decodes terms in the binary format. It is generally a lot faster than `erl_interface` and suitable to use in drivers and port programs.

The functions in ei is provided in library, with some header files. Each function is described in the reference manual.

...

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/hosts, /etc/resolv.conf, etc) in these modes.

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.

`{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.

`{alt_nameserver, IP [,Port]}.` IP = tuple()
Port = integer()
Add address (and port, if other than default) of secondary nameserver.

`{search, Domains}.` Domains = [string()]
Add search domains.

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

`{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 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 millesec) for DNS queries. Default is 2 sec.

`{retry, N}.` N = integer()
Set the number of DNS queries to try before giving up. Default is 3.

`{inet6, Bool}.` Bool = true | false
Tells the system to use IPv6. Default is false.

`{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). 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"]}.
%% read nameserver info from here
{file, resolv, "/etc/resolv.conf"}.
%% 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"'
```

ERTS Reference Manual

Short Summaries

- Command **epmd** [page 47] – Erlang Port Mapper Daemon
- Command **erl** [page 48] – The Erlang Emulator
- Command **erlc** [page 54] – Compiler
- Command **erlsrv** [page 57] – Run the Erlang emulator as a service on Windows NT
- Command **run_erl** [page 62] – Redirect Erlang input and output streams on Solaris
- Command **start** [page 64] – OTP start script example for Unix
- Command **start_erl** [page 65] – Start Erlang for embedded systems on Windows NT
- Command **werl** [page 67] – The Erlang Emulator
- C Library **erl_set_memory_block** [page 68] – Custom memory allocation for Erlang on VxWorks
- C Library **erts_alloc** [page 70] – An Erlang Run-Time System internal memory allocator library.
- Erlang Module **driver_entry** [page 76] – The driver-entry structure used by erlang drivers.
- Erlang Module **erl_driver** [page 79] – API functions for an erlang driver

epmd

The following functions are exported:

- `epmd [-daemon]` Start a name server as a daemon
- `epmd -names` Request the names of the registrered Erlang nodes on this host
- `epmd -kill` Kill the `epmd` process
- `epmd -help` List options

erl

The following functions are exported:

- `erl <arguments>` Start an Erlang runtime system

erlc

The following functions are exported:

- `erlc flags file1.ext file2.ext...` Compile files

erlsrv

The following functions are exported:

- `erlsrv {set | add} <service-name> [<service options>]` Add or modify an Erlang service
- `erlsrv {start | stop | disable | enable} <service-name>` Manipulate the current service status.
- `erlsrv remove <service-name>` Remove the service.
- `erlsrv list [<service-name>]` List all erlang services or all options for one service.
- `erlsrv help` Display a brief help text

run_erl

The following functions are exported:

- `run_erl [-daemon] pipe_dir/ log_dir "exec command [command_arguments]"` Start the Erlang emulator without attached terminal

start

The following functions are exported:

- `start [ data_file ]` This is an example script on how to startup the Erlang system in embedded mode on Unix.

start_erl

The following functions are exported:

- `start_erl [<erl options>] ++ [<start_erl options>]` Start the Erlang emulator with the correct release data

werl

No functions are exported.

erl_set_memory_block

The following functions are exported:

- `int erl_set_memory_block(size_t size, void *ptr, int warn_mixed_malloc, int realloc_always_moves, int use_reclaim, ...)`
  Specify parameters for Erlang internal memory allocation.
- `int erl_memory_show(...)` A utility similar to VxWorks `memShow`, but for the Erlang memory area.
- `int erl_mem_info_get(MEM_PART_STATS *stats)` A utility similar to VxWorks `memPartInfoGet`, but for the Erlang memory area.

erts_alloc

No functions are exported.

driver_entry

The following functions are exported:

- `int init(void)`
  [page 76] Called after loading of driver
- `int start(ErlDrvPort port, char* command)`
  [page 76] Called when port is opened
- `void stop(ErlDrvData drv_data)`
  [page 76] Called when port is closed
- `void output(ErlDrvData drv_data, char *buf, int len)`
  [page 77] Called when port is written to
- `void ready_input(ErlDrvData drv_data, ErlDrvEvent event)`
  [page 77] Called when the driver event for input or output is signaled
- `void ready_output(ErlDrvData drv_data, ErlDrvEvent event)`
  [page 77] Called when the driver event for input or output is signaled
- `char *driver_name`
  [page 77] The name of the driver
- `void finish(void)`
  [page 77] Called just before the dynamic driver is unloaded
- `void *handle`
  [page 77] Reserved, set to `NULL`
- `int control(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen)`
  [page 77] Invoked with `port_control`
- `void timeout(ErlDrvData drv_data)`
  [page 78] Called when timer reaches 0
- `void outputv(ErlDrvData drv_data, ErlIOVec *ev)`
  [page 78] Called when the port is written to
- `void ready_async(ErlDrvData drv_data, ErlDrvThreadData thread_data)`
  [page 78] Called when an asynchronous call has returned
- `int call(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen, unsigned int *flags)`
  [page 78] Synchronous call with term conversion

erl driver

The following functions are exported:

- `ErlDrvBinary`
  [page 80] A driver biary.

- `ErlDrvData`
  [page 80] Driver specific data

- `SysIOVec`
  [page 80] System I/O vector

- `ErlIOVec`
  [page 80] Erlang I/O vector

- `int driver_output(ErlDrvPort port, char *buf, int len)`
  [page 81] Send data from driver to port owner

- `int driver_output2(ErlDrvPort port, char *hbuf, int hlen, char *buf, int len)`
  [page 81] Send data and binary data to port owner

- `int driver_output_binary(ErlDrvPort port, char *hbuf, int hlen, ErlDrvBinary* bin, int offset, int len)`
  [page 81] Send data from a driver binary to port owner

- `int driver_outputv(ErlDrvPort port, char* hbuf, int hlen, ErlIOVec *ev, int skip)`
  [page 81] Send vectorized data to port owner

- `int driver_vec_to_buf(ErlIOVec *ev, char *buf, int len)`
  [page 82] Collect data segments into a buffer

- `int driver_set_timer(ErlDrvPort port, unsigned long time)`
  [page 82] Set a timer to call the driver

- `int driver_cancel_timer(ErlDrvPort port)`
  [page 82] Cancel a previously set timer

- `int driver_read_timer(ErlDrvPort port, unsigned long *time_left)`
  [page 82] Read the time left before timeout

- `int driver_select(ErlDrvPort port, ErlDrvEvent event, int mode, int on)`
  [page 82] Provide an event for having the emulator call the driver

- `void *driver_alloc(size_t size)`
  [page 83] Allocate memory

- `void *driver_realloc(void *ptr, size_t size)`
  [page 83] Resize an allocated memory block

- `void driver_free(void *ptr)`
  [page 83] Free an allocated memory block

- `ErlDrvBinary* driver_alloc_binary(int size)`
  [page 83] Allocate a driver binary

- `ErlDrvBinary* driver_realloc_binary(ErlDrvBinary *bin, int size)`
  [page 83] Resize a driver binary

- `void driver_free_binary(ErlDrvBinary *bin)`
  [page 84] Free a driver binary

- `int driver_enq(ErlDrvPort port, char* buf, int len)`
  [page 84] Enqueue data in the driver queue

- `int driver_pushq(ErlDrvPort port, char* buf, int len)`
  [page 84] Push data at the head of the driver queue

- `int driver_deq(ErlDrvPort port, int size)`
  [page 84] Dequeue data from the head of the driver queue

- `int driver_sizeq(ErlDrvPort port)`
  [page 84] Return the size of the driver queue

- `int driver_enq_bin(ErlDrvPort port, ErlDrvBinary *bin, int offset, int len)`
  [page 84] Enqueue binary in the driver queue

- `int driver_pushq_bin(ErlDrvPort port, ErlDrvBinary *bin, int offset, int len)`
  [page 84] Push binary at the head of the driver queue

- `SysIOVec* driver_peekq(ErlDrvPort port, int *vlen)`
  [page 84] Get the driver queue as a vector

- `int driver_enqv(ErlDrvPort port, ErlIOVec *ev, int skip)`
  [page 85] Enqueue vector in the driver queue

- `int driver_pushqv(ErlDrvPort port, ErlIOVec *ev, int skip)`
  [page 85] Push vector at the head of the driver queue

- `void add_driver_entry(ErlDrvEntry *de)`
  [page 85] Add a driver entry

- `int remove_driver_entry(ErlDrvEntry *de)`
  [page 85] Remove a driver entry

- `char* erl_errno_id(int error)`
  [page 85] Get erlang error atom name from error number

- `void set_busy_port(ErlDrvPort port, int on)`
  [page 85] Signal or unsignal port as busy

- `void set_port_control_flags(ErlDrvPort port, int flags)`
  [page 85] Set flags on how to handle control entry function

- `int driver_failure_eof(ErlDrvPort port)`
  [page 85] Fail with EOF

- `int driver_failure_atom(ErlDrvPort port, char *string)`
  [page 86] Fail with error

- `int driver_failure_posix(ErlDrvPort port, int error)`
  [page 86] Fail with error

- `int driver_failure(ErlDrvPort port, int error)`
  [page 86] Fail with error

- `ErlDriverTerm driver_connected(ErlDrvPort port)`
  [page 86] Return the port owner process

- `ErlDriverTerm driver_caller(ErlDrvPort port)`
  [page 86] Return the process making the driver call

- `int driver_output_term(ErlDrvPort port, ErlDriverTerm* term, int n)`
  [page 86] Send term data from driver to port owner

- `ErlDriverTerm driver_mk_atom(char* string)`
  [page 88] Make an atom from a name

- `ErlDriverTerm driver_mk_port(ErlDrvPort port)`
  [page 88] Make a erlang term port from a port

- `int driver_send_term(ErlDrvPort port, ErlDriverTerm receiver, ErlDriverTerm* term, int n)`
  [page 88] Send term data to other process than port owner process

- `long driver_async (ErlDrvPort port, unsigned int* key, void (*async_invoke)(void*), void* async_data, void (*async_free)(void*))`
  [page 88] Perform an asynchronous call within a driver

- `int driver_async_cancel(long id)`
  [page 88] Cancel an asynchronous call

- `int driver_send_term(ErlDrvPort port, ErlDriverTerm receiver, ErlDriverTerm* term, int n)`

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.

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
*&ld;TABs&gt;/var/log/epmd.log
```

where <TABs> are real tab characters. 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 [page 51].

  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 [page 49] 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)].

`-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 [page 54] 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 neither 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) [page 47].

   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)].

-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.

-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` Sets the number of threads in async thread pool, valid range is 0..256. 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.

+h `Size` Sets the default 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.

+M`Flag Value` Memory allocator specific flags, see erts_alloc(3) [page 70] 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.

+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.

> **Note:**
> The +m, +t, and +T flags have changed to, respectively, +MYm [page 73], +MYtt [page 73], and +MYtp [page 73]. The +d, and +S* flags have been removed. See erts_alloc(3) [page 70] 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_FLAGS The value of this environment variable will be appended to the command line for `erl`.

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) [page 47], erts_alloc(3) [page 70]

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.

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:

1. `"."`, the current working directory of the file server;
2. the base name of the compiled file;
3. 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*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.

– 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 fourth argument of the `yecc:yecc/4` function).
Supported options: -o, -v, -I (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), erl_compile(3), compile(3), yecc(3), snmp(3)

erlsrv

Command

This utility is specific to Windows NT It allows Erlang emulators to run as services on the NT 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 NT services applet in a manner similar to other services.

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 NT.

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 logfile 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 NT 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. If no `DebugType` is specified (`none`), the output of the Erlang shell is discarded.

- `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`).

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 [page 60]).

-**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 NT 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.

-**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.

`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 [page 58]).

SEE ALSO

start_erl(1), release_handler(3)

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)

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 (includeing 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)

werl

Command

On Windows, the preferred way to start the Erlang system is:

`werl <script-flags> <user-flags>`

This will start Erlang in its own window, which is nice for interactive use (command-line editing will work and there are scrollbars). All flags except the `-oldshell` flag work as in `erl`.

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 there 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.

erl_set_memory_block

C Module

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.

erts_alloc

C Module

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.

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 [page 73]. 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 [page 74]) 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 [page 73] parameter.

Sizes of multiblock carriers allocated via `mseg_alloc` are decided based on the values of the largest multiblock carrier size (lmbcs [page 73]), the smallest multiblock carrier size (smbcs [page 74]), and the multiblock carrier growth stages (mbcgs [page 74]) 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 [page 74]) 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 [page 73] 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 [page 73] 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.

NOTE: Do *not* use this strategy for other allocators than `temp_alloc`. If you do, you *will* run into trouble.

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 [page 48]) 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`
- 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 specific allocators based on
alloc_util (i.e. <S> is either B, D, E, H, L, S, or T):

+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 [page 71] 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 [page 74].

+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 [page
  71] 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 [page 71]
  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>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 [page 74] 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 [page 73].

+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 [page 71] in "the alloc_util framework" section.

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 [page 50].

+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` min disables all allocators that can be disabled. `max` enables all allocators. `r9c` configures all allocators as they were configured in the OTP R9C release. The `r9c` switch will eventually be removed.

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. See also the documentation for [erlang:system_info/1] in the [erlang(3)] documentation.

> **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).

See Also

erl(1) [page 48], [instrument(3)], [erlang(3)]

driver_entry

Erlang Module

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.

All functions are function pointers. The `driver_init` function returns a pointer to the `driver_entry` structure. The name in the structure must correspond to the name of the driver, and the driver library file name (without file extension).

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 nessecary, since the emulator must be responsive.

Exports

`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 not used, it's still around only for historical reasons. It should be `NULL`. Don't use it.

`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 as a driver binary, 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 pointer to a buffer of `rlen` bytes, which can be used to return data. Data is returned depending of the port control flags (those that are set with `set_port_control_flags` [page 85]). If the flag is set to `PORT_CONTROL_FLAG_BINARY`, then `rbuf` should point to a driver binary or be NULL. Note that this binary must be freed. If `rbuf` is set to NULL, an empty list will be returned.

If the flag is set to 0, rbuf points to a char* containing data, that is returned as a list of integers. 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 activeated 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 [page 84] 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 [page 88]. 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.

See Also

erl_driver(3), erl_ddll(3), kernel(3), erlang(3)

erl_driver

Erlang Module

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 [page 76]).

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.)

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 [page 78] entry function after a specified time. Only one timer is available for each driver instance.

Queue handling Every driver 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.

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 thet 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 asynchonous function calls, using a thread pool provided by erlang. There is also a select call, that can be used for asynchronous drivers.

Adding / remove drivers A driver can add and later remove drivers.

Exports

ErlDrvBinary

Types:

- int orig_size
- int refc
- char orig_bytes[]

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 `refc` field 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)`.

Some driver calls, such as `driver_enq_binary`, increments the driver ref-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, 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 [page 78] 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 ref-count in the `refc` field should be incremented, and the binary can later be freed in the stop [page 76] 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, shouldn't be changed by the driver.

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

Types:

- int vsize
- int size

- SysIOVec* iov
- ErlDrvBinary** binv

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 [page 78] driver call-back. Also, the driver queue is an `ErlIOVec`.

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` [page 78] driver entry function. You can also make them yourself, if you want to send several `ErlDriverBinary` 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 [page 76] 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_select(ErlDrvPort port, ErlDrvEvent event, int mode, int on)`

The `driver_select` is used by the driver to provide the emulator with an event to check for. This enables the emulator to call the driver when something has happened asynchronously.

The `event` parameter is used in the emulator cycle in a `select` call. If the event is set then the driver is called. The `mode` parameter can be either `ON_READ` or `ON_WRITE`, and specifies whether ready_output [page 77] or ready_input [page 77] will be called when the event is fired. Note that this is just a convention, they don't have to read or write anything.

The `on` parameter should be `1` for adding the event and `0` for removing it.

On unix systems, the function `select` is used. The `event` must be a socket or pipe (or other object that `select` can use).

On windows, the Win32 API function `WaitForMultipleObjects` is used. This places other restriction on the `event`. Refer to the Win32 SDK documentation.

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. Every `driver_alloc` call must have a corresponding `driver_free`.

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 ofthen a wrapper for `realloc`.)

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.

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 shouldn't be altered. Every allocated binary should be freed.

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.

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.

`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. Every call to `driver_alloc_binary` should have a matching call to `driver_free_binary`.

`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.

`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.

`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 dealloced.
>
> The return value is 0.

`int driver_sizeq(ErlDrvPort port)`

> This function returns the number of bytes currently in the driver queue.

`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.
>
> 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.
>
> 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 `SysIOVecs`. 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`.

`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.

`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.

`void add_driver_entry(ErlDrvEntry *de)`

This function adds a driver entry to the list of drivers known by erlang. The init [page 76] function of the `de` parameter is called.

`int remove_driver_entry(ErlDrvEntry *de)`

This function removes a driver entry `de` previously added with `add_driver_entry`.

`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.

`void set_port_control_flags(ErlDrvPort port, int flags)`

This function sets flags for how the control [page 77] 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 return from `control` is sent to the port owner process.

`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 kepp 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.

```
ErlDriverTerm driver_connected(ErlDrvPort port)
```

This function returns the port owner process.

```
ErlDriverTerm driver_caller(ErlDrvPort port)
```

This function returns the process that made the current call to the driver. This can be used with `driver_send_term` to send back data to the caller. (This is the process that called one of `erlang:send/2`, `erlang:port_command/2` or `erlang:port_control/3`).

```
int driver_output_term(ErlDrvPort port, ErlDriverTerm* term, int n)
```

This functions sends data in the special driver term format. This is a fast way to deliver term data to 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 `ErlDriverTerm`, 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          None
ERL_DRV_ATOM         driver_mk_atom(string)
ERL_DRV_INT          int
ERL_DRV_PORT         driver_mk_port(ix)
ERL_DRV_BINARY       ErlDriverBinary*, int len, int offset
ERL_DRV_STRING       char*, int len
ERL_DRV_TUPLE        int size
ERL_DRV_LIST         int size
ERL_DRV_PID          driver_connected() or driver_caller()
ERL_DRV_STRING_CONS  char*, int len
ERL_DRV_FLOAT        double*
```

To build the tuple {tcp, Port, [100 | Binary]}, the following call could be made.

```
ErlDriverBinary* bin = ...
ErlDriverPort port = ...
ErlDriverTerm 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: */
ErlDriverTerm spec[] = {
    ERL_DRV_ATOM, driver_mk_atom("x"),
    ERL_DRV_STRING, (ErlDriverTerm)"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: */
ErlDriverTerm spec[] = {
    ERL_DRV_NIL,          /* with STRING_CONS, the tail comes first */
    ERL_DRV_STRING_CONS, (ErlDriverTerm)"123", 3,
    ERL_DRV_STRING_CONS, (ErlDriverTerm)"abc", 3,
};
driver_output_term(port, spec, sizeof(spec) / sizeof(spec[0]));
```

`ErlDriverTerm 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.

`ErlDriverTerm 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, ErlDriverTerm receiver, ErlDriverTerm* 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 [page 86].

`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.

> Normally, erlang is started without a thread pool. A start argument to the emulator, specifies how many threads that should be available (e.g. `+A 5`, gives five extra driver threads). If no thread pool is available, the call is made synchronously, in the emulator thread.

> 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:

> ```
> r = driver_async(myPort, (unsigned char*)&myPort, myData, myFunc);
> ```

> 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 [page 78] 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`.

`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.

See Also

driver_entry(3), erl_ddll(3), erlang(3)
An Alternative Distribution Driver (ERTS User's Guide Ch. 3)

List of Tables

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.

Erlang Run-Time System Application (ERTS)