Erlang Run-Time System Application (ERTS)

version 5.0

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

Contents

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 (preferably the parameters to a function as used in the `erlang:trace_pattern/2` BIF). 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).

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'`.

Grammar

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

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

- ActionFunction ::= `set_seq_token` | `message` | `return_trace` | `process_dump` | `enable_trace` | `disable_trace` | `display` | `caller`

Function descriptions

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(<record_type>, size)`, like in `{is_record, '$1', rectype, record_info(rectype, size)}`.

'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.

andthen: 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'`.

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.

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.

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.

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

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.

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.

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.

Note that if a "technically built in function" (i.e. a function not written in Erlang) is traced, the `caller` function will always 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.

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.

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) 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

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.
- 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.

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.

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.

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 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 simultaneous atoms in the system, or too many simultaneous ets tables). Usually the emulator or the operating system can be reconfigured to avoid the crash, why interpreting the crash dump correctly is important.

Reasons for crash dumps

The reason for the dump is noted in the beginning of the file as `Slogan: <reason>` (the word Slogan has historical reasons). If the system is halted by the bif `erlang:halt/1`, the slogan is exactly the parameter passed to the bif (a string), 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.

- "Can't allocate *N* bytes of memory" - The system has run out of memory. The number `N` indicates the amount of memory needed (in bytes) , which could give some hint of what the problem is. If `N` is very large, it could be that an Erlang process consumes vast amounts of memory, possibly due to an error in the Erlang code.
- "Can't reallocate *N* bytes of memory" - Same as above.
- "Can't allocate *Something*" - Same as above.
- "Got unusable memory block *Address*, size *N*" - The emulator has reached the 1GB limit of the Erlang virtual memory space. Something consumes huge amounts of memory, probably an error in the Erlang code.
- "Unexpected op code *N*" - Error in compiled code, `beam` file damaged or error in 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 pass 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`). That the application controller has shut down can be for various 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 has 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". 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 one's 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 quite useful for tracking down the problem.

Process information

After the general information in the crash dump (the date, slogan and version information) follows a listing of each living Erlang process in the system. The process information for one process may look like this (except for the line numbers of course):

```
(1)  <0.2.0> Waiting. Registered as: erl_prim_loader
(2)  Spawned as: erl_prim_loader:start_it/4
(3)  Message buffer data: 262 words
(4)  Link list: [<0.0.0>,<0,1>]
(5)  Dictionary: [{fake, entry}]
(6)  Reductions 2194 stack+heap 987 old_heap_sz=987
(7)  Heap unused=85 OldHeap unused=987
(8)  Stack dump:
(9)  130ef8     Blank a
(10)  130ef4    Blank a
(11) 130ef0     Blank a
(12) 130eec     <0.1.0>
(13) 130ee8     {state,[],none,get_from_port_efile,stop_port,exit_port,\
     <0,1>,infinity,dummy_in_handler}
(14) 130ee4     ["/ldisk/r6a_test/kernel_test","/ldisk/r6a_dev/lib/\n     kernel-2.4/ebin","/ldisk
(15) 130ee0     Continuation pointer b754c,
(16) i = 0x125750, cp = 0xb754c, arity = 0, 125710: erl_prim_loader:loop/3,
```

Each line of the output should be interpreted as follows:

- *(1)* - The process id (`<0.2.0>`), the state of the process (`Waiting`) and the registered name of the process, if any (`erl_prim_loader`). The state of the process 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 it's way to exit.
  - *Process is garbing, limited information.* - 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's trying to write to a busy port.
- *(2)* - The entry point of the process, i. e. what function was referenced in the `spawn` or `spawn_link` call that started the process.
- *(3)* - Size of fragmented heap data (incorrectly called message buffers). This is data either created by messages being sent to the process or by the Erlang bif's. This amount depends on so many things, that this field is utterly uninteresting.
- *(4)* - 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.
- *(5)* - The contents of the process dictionary (the `put/2` and `get/1` thing).
- *(6)* - The number of reductions consumed by the process, the size of the stack and heap (they share memory segment) and the size of the "old heap". This "old heap" may require some explanation. 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 has survived two garbage collections. The assumption (which is almost always correct) is that data that survives 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' allocated memory.
- *(7)* - The amount of unused memory on each heap. This information is usually useless.
- *(8) - (15)* - A dump of the Erlang process stack. Most of the live data (i.e. variables currently in use) are placed on the stack, why 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 `(13)`. The stack also contains other things, like "Continuation pointer", "Blank" "Frame", "Catch" and such. Those fields are only interesting for developers who understand the internal structure of the beam emulator.
- *(16)* - Miscellaneous information abut the process state:
  - *i* - The current instruction pointer, only interesting for runtime system developers.
  - *cp* - The current continuation pointer, i.e. the return address for the current call. Usually useless for other than runtime system developers.
  - *arity* - Number of live argument registers, quite uninteresting.
  - The function into which *i* points - This is the current function of the process. This may be followed by the function into which *cp* points, which is the function calling the current function. The argument registers, if any living, may follow. These may contain the arguments of the function if they are not yet moved to the stack.

  Usually the current function is the only useful information on this line.

Internal table information

This section mostly contains information for runtime system developers. What can be of interest is the following fields:

- *Hash Table(atom_tab)* - The number of objects in the atom table, which is indicated by the field `objs(`*N*`)`, is the number of atoms present in the system at the time of the crash. 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.
- *Hash Table(module_code)* - The field `objs(`*N*`)` indicates the number of loaded modules in the system.
- *Allocated binary* `N` - This number indicates how many bytes are allocated to binaries (the binary data-type) for the whole system. Binaries allocated directly on process heaps (small binaries) are not counted for here.

The rest of the information is only of interest for runtime system developers.

ETS tables

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

- Table *Number*(with name)*Name* - The identifier and the name of the table.
- Owner *Pid* - The process owning the table.
- Buckets: *N* | Ordered set (AVL tree), Elements: *N* - The most interesting here is that it indicates if the table is a `ordered_set` or not.
- Table's got *N* objects - the number of objects in the table
- Table's got *N* words of active data - The number of words (usually 4 bytes/word) allocated to data in the table.

Timers

This section contains information about all the timers started with the bif `erlang:start_timer/3`. Each line includes the message to be sent, the pid to receive the message and how many milliseconds there was left until the message would have been sent.

Loaded module information

This is a list of all loaded modules, together with the memory usage of each module, in bytes. Note that loaded code is usually larger than the packed format in the beam files.

At the end of the list, 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.

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.

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’s 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.

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).

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 erlangs 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <fcntl.h>

#define HAVE_UIO_H
#include "erl_driver.h"

/*
```

```
(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:

```
typedef enum {
    portTypeUnknown,      /* An uninitialized port */
    portTypeListener,     /* A listening port/socket */
    portTypeAcceptor,     /* An intermidiate stage when accepting
                             on a listen port */
    portTypeConnector,    /* An intermediate stage when connecting */
    portTypeCommand,      /* A connected open port in command mode */
    portTypeIntermediate, /* A connected open port in special
                             half active mode */
    portTypeData          /* A connectec open port in data mode */
} 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.

```
typedef unsigned char Byte;
typedef unsigned int Word;

typedef struct uds_data {
    int fd;                   /* File descriptor */
    ErlDrvPort port;          /* The port identifier */
    int lockfd;               /* The file descriptor for a lock file in
                                 case of listen sockets */
    Byte creation;            /* The creation serial derived from the
                                 lockfile */
    PortType type;            /* Type of port */
    char *name;               /* Short name of socket for unlink */
    Word sent;                /* Bytes sent */
    Word received;            /* Bytes received */
    struct uds_data *partner; /* The partner in an accept/listen pair */
    struct uds_data *next;    /* Next structure in list */
    /* The input buffer and it's data */
    int buffer_size;          /* The allocated size of the input buffer */
    int buffer_pos;           /* Current position in input buffer */
    int header_pos;           /* Where the current header is in the
                                 input buffer */
    Byte *buffer;             /* The actual input buffer */
} 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, detais 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.

```
/* Beginning of linked list of ports */
static UdsData *first_data;


DRIVER_INIT(uds_drv)
{
    first_data = NULL;
    return &uds_driver_entry;
}
```

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.

```
static ErlDrvData uds_start(ErlDrvPort port, char *buff)
{
    UdsData *ud;

    ud = ALLOC(sizeof(UdsData));
    ud->fd = -1;
    ud->lockfd = -1;
    ud->creation = 0;
    ud->port = port;
    ud->type = portTypeUnknown;
    ud->name = NULL;
    ud->buffer_size = 0;
    ud->buffer_pos = 0;
    ud->header_pos = 0;
    ud->buffer = NULL;
    ud->sent = 0;
    ud->received = 0;
    ud->partner = NULL;
    ud->next = first_data;
    first_data = ud;

    return((ErlDrvData) ud);
}
```

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:

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

    if (ud->type == portTypeData || ud->type == portTypeIntermediate) {
        DEBUGF(("Passive do_send %d",bufflen));
        do_send(ud, buff + 1, bufflen - 1); /* XXX */
        return;
    }
    if (bufflen == 0) {
        return;
    }
    switch (*buff) {
    case 'L':
        if (ud->type != portTypeUnknown) {
            driver_failure_posix(ud->port, ENOTSUP);
            return;
        }
        uds_command_listen(ud,buff,bufflen);
        return;
    case 'A':
        if (ud->type != portTypeUnknown) {
            driver_failure_posix(ud->port, ENOTSUP);
            return;
        }
        uds_command_accept(ud,buff,bufflen);
        return;
    case 'C':
        if (ud->type != portTypeUnknown) {
            driver_failure_posix(ud->port, ENOTSUP);
            return;
        }
        uds_command_connect(ud,buff,bufflen);
        return;
    case 'S':
        if (ud->type != portTypeCommand) {
            driver_failure_posix(ud->port, ENOTSUP);
            return;
        }
        do_send(ud, buff + 1, bufflen - 1);
        return;
    case 'R':
        if (ud->type != portTypeCommand) {
            driver_failure_posix(ud->port, ENOTSUP);
            return;
        }
        do_recv(ud);
        return;
    default:
        return;
    }
}
```

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 OTP-R7.

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:

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

    if (ud->type == portTypeListener) {
        UdsData *ad = ud->partner;
        struct sockaddr_un peer;
        int pl = sizeof(struct sockaddr_un);
        int fd;

        if ((fd = accept(ud->fd, (struct sockaddr *) &peer, &pl)) < 0) {
            if (errno != EWOULDBLOCK) {
                driver_failure_posix(ud->port, errno);
                return;
            }
            return;
        }
        SET_NONBLOCKING(fd);
        ad->fd = fd;
        ad->partner = NULL;
        ad->type = portTypeCommand;
        ud->partner = NULL;
        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
        driver_output(ad->port, "Aok",3);
        return;
    }
    do_recv(ud);
}
```

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:

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

    put_packet_length(header, bufflen);
    iov[0].iov_base = (char *) header;
    iov[0].iov_len = 4;
    iov[1].iov_base = buff;
    iov[1].iov_len = bufflen;
    eio.iov = iov;
    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:

```
static int send_out_queue(UdsData *ud)
{
    for(;;) {
        int vlen;
        SysIOVec *tmp = driver_peekq(ud->port, &vlen);
        int wrote;
        if (tmp == NULL) {
            driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_WRITE, 0);
            if (ud->type == portTypeCommand) {
                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:

```
static void uds_output(ErlDrvData handle, ErlDrvEvent event)
{
   UdsData *ud = (UdsData *) handle;
   if (ud->type == portTypeConnector) {
       ud->type = portTypeCommand;
       driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_WRITE, 0);
       driver_output(ud->port, "Cok",3);
       return;
   }
   send_out_queue(ud);
}
```

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`:

```
static int uds_control(ErlDrvData handle, unsigned int command,
                       char* buf, int count, char** res, int res_size)
{
/* Local macro to ensure large enough buffer. */
#define ENSURE(N)                               \n( 6)    do {

    UdsData *ud = (UdsData *) handle;

    switch (command) {
    case 'S':
        {
            ENSURE(13);
            **res = 0;
            put_packet_length((*res) + 1, ud->received);
            put_packet_length((*res) + 5, ud->sent);
            put_packet_length((*res) + 9, driver_sizeq(ud->port));
            return 13;
        }
    case 'C':
        if (ud->type < portTypeCommand) {
            return report_control_error(res, res_size, "einval");
        }
        ud->type = portTypeCommand;
        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
        ENSURE(1);
        **res = 0;
        return 1;
    case 'I':
        if (ud->type < portTypeCommand) {
            return report_control_error(res, res_size, "einval");
        }
        ud->type = portTypeIntermediate;
        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
        ENSURE(1);
        **res = 0;
        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.

The erlang interface to the driver

Not written yet

The distribution module

The dist_util module

Not written yet

The uds_dist callback module

Not written yet

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 \n      -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 \n      -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 preferrably not include the name of the node.

1.4 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*.

Normal Mode

In normal mode keystrokes from the user are collected and interpreted by `tty`. Most of the *emacs* line editing commands are supported. The following is a complete list of the supported line editing commands.
Note: The notation `C-a` means pressing the control key and the letter `a` simultaneously. `M-f` means pressing the `ESC` key followed by the letter `f`.

Key Sequence	*Function*
C-a	Beginning of line
C-b	Backward character
M-b	Backward word
C-d	Delete character
M-d	Delete word
C-e	End of line
C-f	Forward character
M-f	Forward word
C-g	Enter shell break mode
C-k	Kill line
C-l	Redraw line
C-n	Fetch next line from the history buffer
C-p	Fetch previous line from the history buffer

continued ...

... continued

C-t	Transpose characters
C-y	Insert previously killed text

Table 1.2: tty text editing

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

ERTS Reference Manual

Short Summaries

- Command **epmd** [page 36] – Erlang Port Mapper Daemon
- Command **erl** [page 38] – The Erlang Emulator
- Command **erlc** [page 42] – Compiler
- Command **erlsrv** [page 45] – Run the Erlang emulator as a service on Windows NT(R)
- Command **run_erl** [page 50] – Redirect Erlang input and output streams on Solaris(R)
- Command **start** [page 51] – OTP start script example for Unix
- Command **start_erl** [page 52] – Start Erlang for embedded systems on Windows NT(R)
- Command **werl** [page 54] – The Erlang Emulator
- C Library **erl_set_memory_block** [page 55] – Custom memory allocation for Erlang on VxWorks(R)

epmd

The following functions are exported:

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

erl

The following functions are exported:

- `erl <arguments>` Starts the Erlang system

erlc

The following functions are exported:

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

erlsrv

The following functions are exported:

- `erlsrv {set | add} <service-name> [<service options>]` Adds or modifies an Erlang service
- `erlsrv {start | stop | disable | enable} <service-name>` Manipulates the current service status.
- `erlsrv remove <service-name>` Removes the service.
- `erlsrv list [<service-name>]` Lists all erlang services or all options for one service.
- `erlsrv help` Displays a brief help text

run_erl

The following functions are exported:

- `run_erl pipe_dir/ log_dir "exec command [command_arguments]"` Start the Erlang emulator with the correct release data

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, ...)` Specifies 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.

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 the Erlang runtime system. The exact details (e.g. whether `erl` is a script or a program and which other programs it calls) are system-dependent.

Windows 95/98/2000/NT users will probably want to use the `werl` program instead, which run 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 the Erlang system.

Any argument starting with a plus sign (+) is always interpreted as a system flag (described below), regardless of where it occurs on the command line.

Arguments starting with a hyphen (-) are the start of a flag. A flag includes all following arguments up to the next argument starting with a hyphen.

Example:

```
erl -sname arne -myflag 1 -s mod func arg
```

Here `-sname arne` is a flag and so are `-myflag 1` and `-s mod func arg`. Note that these flags are treated differently. `-sname arne` is interpreted by the OTP system, but it still included in the list of flags returned by `init:get_arguments/0`. `-s mod func arg` is also treated specially and it is not included in the return value for `init:get_arguments/0`. Finally, `-myflag 1` is not interpreted by the OTP system in any way, but it is included in `init:get_arguments/0`.

Plain arguments are not interpreted in any way. They can be retrieved using `init:get_args/0`. Plain arguments can occur in two places: Before the first flag argument on the command line, or after a `--` argument.

Flags

The following flags are supported:

– Any arguments `--` following will not be interpreted in any way. They can be retrieved by `init:get_args/0`. The exception is arguments starting with a `+`, which will be interpreted as system flags (see below).

-AppName Key Value Overrides the `Key` configuration parameter of the `AppName` application. See application(3). This type of flag can also be retrieved using the `init` module.

-boot File Specifies the name of the boot script, `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 `<ERL_INSTALL_DIR>/bin` directories.

If this flag is omitted, the `<ERL_INSTALL_DIR>/bin/start.boot` boot script is used.

-boot_var Var Directory [Var Directory] If the boot script used contains another path variable than $ROOT, this variable must have a value assigned in order to start the system. A boot variable is used if user applications have been installed in another location than underneath the `<ERL_INSTALL_DIR>/lib` directory. `$Var` is expanded to `Directory` in the boot script.

-compile mod1 mod2 .... Makes the Erlang system compile `mod1.erl mod2.erl ....` and then terminate (with non-zero exit code if the compilation of some file didn't succeed). Implies `-noinput`. Not recommended - use erlc(1) instead.

-config Config Reads the `Config`.config configuration file in order to configure the system. See 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 Obsolete flag without any effect and common misspelling for `-setcookie`. Use `-setcookie Cookie` option if want to override the default cookie.

-detached Starts the Erlang 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` of the Erlang system. For example:

```
% erl -env DISPLAY gin:0
```

In this example, an Erlang system is started with the DISPLAY environment variable set to the value `gin:0`.

-heart Starts heart beat monitoring of the Erlang system. See heart(3).

-hosts Hosts Specifies the IP addresses for the hosts on which an Erlang boot servers are running. This flag is mandatory if the `-loader inet` flag is present. On each host, there must be one Erlang node running, on which the `boot_server` must be started.

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 an broadcast address (preferably limited to the local network) is.

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

-instr Selects an instrumented Erlang system (virtual machine) to run, instead of the ordinary one. When running an instrumented system, some resource usage data can be obtained and analysed using the module `instrument`. Functionally, it behaves exactly like an ordinary Erlang system.

-loader Loader Specifies the name of the loader used to load Erlang modules into the system. See erl_prim_loader(3). `Loader` can be `efile` (use the local file system), or `inet` (load using the `boot_server` on another Erlang node). If `Loader` is something else, the user supplied `Loader` port program is started.

If the `-loader` flag is omitted `efile` is assumed.

-make Makes the Erlang system invoke `make:all()` in the current work 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 Mode The mode flag indicates if the system will load code automatically at runtime, or if all code is loaded during system initialization. `Mode` can be either `interactive` to allow automatic code loading, or `embedded` to load all code during start-up. See code(3).

-name Name Makes the node a distributed node. This flag invokes all network servers necessary for a node to become distributed. See net_kernel(3).

The name of the node will be `Name@Host`, where `Host` is the fully qualified host name of the current host. This flag also ensures that `epmd` runs on the current host before Erlang is started. See epmd(1).

-noinput Ensures that the Erlang system never tries to read any input. Implies `-noshell`.

-noshell Starts an Erlang system with no shell at all. This flag makes it possible to have the Erlang system as a component in a series of UNIX pipes.

-nostick Disables the sticky directory facility of the `code server`. See code(3).

-oldshell Invokes the old Erlang shell from Erlang release 3.3. The old shell can still be used.

-pa Directories Adds the directories `Directories` to the head of the search path of the code server, as if `code:add_pathsa/1` was called. See code(3).

-pz Directories Adds the directories `Directories` to the end of the search path of the code server, as if `code:add_pathsa/1` was called. See code(3).

-s Mod [Fun [Args]] Passes the `-s` flag to the `init:boot()` routine. See init(3).

-setcookie Cookie Sets the magic cookie of the current node to `Cookie`. As `erlang:set_cookie(node(),Cookie)` is used, all other nodes will also be assumed to have their cookies set to `Cookie`. In this way, several nodes can share one magic cookie. Erlang magic cookies are explained in auth(3).

-sname Name This is the same as the `-name` flag, with the exception that the host name portion of the node name will not be fully qualified. The following command is used do start Erlang at the host with the name `gin.eua.ericsson.se`

```
% erl -sname klacke
Eshell V4.7 (abort with ^G)
(klacke@gin)1>
```

Only the host name portion of the node name will be relevant. 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 systems running with the `-sname` flag and those running with the `-name` flag, as node names must be unique in distributed Erlang systems.

-version Makes the system print out its version number.

All these flags are processed during the start-up of the Erlang kernel servers and before any user processes are started. All flags are passed to `init:boot(Args)`. See init(3). All additional flags passed to the script will be passed to `init:boot/2` as well, and they can be accessed using the `init` module.

System Flags

The `erl` script invokes the code for the Erlang virtual machine. This program supports the following flags:

+A size Sets the pool size for device driver threads. Default is 0.

+B De-activates the break handler for ^C and ^\ .

+h size Sets the default heap size of processes to the size `size`.

+l Displays info while loading code.

+P Number Sets the total number of processes for this system. The `Number` must be in the range [15,32768].

+s size Sets the default stack size for Erlang processes to the size `size`.

+v Verbose

+V Prints the version of Erlang at start-up.

Example:

```
% erl -name foo +B +l
```

In this example, a distributed node is started with the break handler turned off and a lot of info is displayed while the code is loading.

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)

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 file in the specified directory. If not given, the compiler assumes that include files are located in the current working directory.

-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 Enables warning messages. Without this switch, only errors will be reported.

-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.

-ilroot ***directory*** Defines the root directory to be used for `include_lib` directives in the Erlang compiler. Defaults to the library directory of the emulator where the compiler is run.

-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: -ilroot, -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.

.h A interface definition for IG (Interface Generator). It generates C and Erlang files.

Supported options: -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(R) 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(R) 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(R).

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(R) 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 48]).

-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(R) 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 46]).

SEE ALSO

start_erl(1), release_handler(3)

run_erl (Command)

This describes the `run_erl` program specific to Solaris. 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 pipe_dir/ log_dir "exec command [command_arguments]"`

The `run_erl` program arguments are:

pipe_dir This is where to put the named pipe, usually `/tmp/`.

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 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`.

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(R) Visual C++(R). 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(R). 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 it's 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 95/NT, 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`.

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.

List of Tables

Index

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