NAME

escript − Erlang scripting support

DESCRIPTION

escript provides support for running short Erlang programs without having to compile them first and an easy way to retrieve the command line arguments.

EXPORTS

script-name script-arg1 script-arg2...
escript escript-flags script-name script-arg1 script-arg2...

escript runs a script written in Erlang.

Here follows an example.

```
$ cat factorial
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -smp enable -sname factorial -mnesia debug verbose
main([String]) ->
  try
    N = list_to_integer(String),
    F = fac(N),
    io:format("factorial ~w = ~w\n", [N,F])
  catch
    _:_ ->
      usage()
  end;
main(_) ->
  usage().

usage() ->
  io:format("usage: factorial integer\n"),
  halt(1).

fac(0) -> 1;
fac(N) -> N * fac(N-1).
$ factorial 5
factorial 5 = 120
$ factorial
usage: factorial integer
$ factorial five
usage: factorial integer
```

The header of the Erlang script in the example differs from a normal Erlang module. The first line is intended to be the interpreter line, which invokes *escript*. However if you invoke the *escript* like this

```
$ escript factorial 5
```

the contents of the first line does not matter, but it cannot contain Erlang code as it will be ignored.

The second line in the example, contains an optional directive to the *Emacs* editor which causes it to enter the major mode for editing Erlang source files. If the directive is present it must be located on the second line.

On the third line (or second line depending on the presence of the Emacs directive), it is possible to give arguments to the emulator, such as

```
%%! -smp enable -sname factorial -mnesia debug verbose
```

Such an argument line must start with *%%!* and the rest of the line will interpreted as arguments to the emulator.

If you know the location of the *escript* executable, the first line can directly give the path to *escript*. For instance:

#!/usr/local/bin/escript

As any other kind of scripts, Erlang scripts will not work on Unix platforms if the execution bit for the script file is not set. (Use *chmod +x script-name* to turn on the execution bit.)

The rest of the Erlang script file may either contain Erlang *source code*, an *inlined beam file* or an *inlined archive file*.

An Erlang script file must always contain the function *main/1*. When the script is run, the *main/1* function will be called with a list of strings representing the arguments given to the script (not changed or interpreted in any way).

If the *main/1* function in the script returns successfully, the exit status for the script will be 0. If an exception is generated during execution, a short message will be printed and the script terminated with exit status 127.

To return your own non-zero exit code, call *halt(ExitCode)*; for instance:

halt(1).

Call **escript:script_name()** from your to script to retrieve the pathname of the script (the pathname is usually, but not always, absolute).

If the file contains source code (as in the example above), it will be processed by the preprocessor *epp*. This means that you for example may use pre-defined macros (such as *?MODULE*) as well as include directives like the *-include_lib* directive. For instance, use

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

to include the record definitions for the records used by the *file:read_link_info/1* function.

The script will be checked for syntactic and semantic correctness before being run. If there are warnings (such as unused variables), they will be printed and the script will still be run. If there are errors, they will be printed and the script will not be run and its exit status will be 127.

Both the module declaration and the export declaration of the *main/1* function are optional.

By default, the script will be interpreted. You can force it to be compiled by including the following line somewhere in the script file:

-mode(compile).

Execution of interpreted code is slower than compiled code. If much of the execution takes place in interpreted code it may be worthwhile to compile it, even though the compilation itself will take a little while. It is also possible to supply *native* instead of compile, this will compile the script using the native flag, again depending on the characteristics of the escript this could or could not be worth while.

As mentioned earlier, it is possible to have a script which contains precompiled *beam* code. In a precompiled script, the interpretation of the script header is exactly the same as in a script containing source code. That means that you can make a *beam* file executable by prepending the file with the lines starting with *#!* and *%%!* mentioned above. In a precompiled script, the function *main/1* must be exported.

As yet another option it is possible to have an entire Erlang archive in the script. In a archive script, the interpretation of the script header is exactly the same as in a script containing source code. That means that you can make an archive file executable by prepending the file with the lines starting with *#!* and *%%!* mentioned above. In an archive script, the function *main/1* must be exported. By default the *main/1* function in the module with the same name as the basename of the *escript* file will be invoked. This behavior can be overridden by setting the flag *-escript main Module* as one of the emulator flags. The *Module* must be the name of a module which has an exported *main/1* function. See **code(3erl)** for more information about archives and code loading.

In many cases it is very convenient to have a header in the escript, especially on Unix platforms. But the header is in fact optional. This means that you directly can "execute" an Erlang module, beam file or archive file without adding any header to them. But then you have to invoke the script like this:

```
$ escript factorial.erl 5
factorial 5 = 120
$ escript factorial.beam 5
factorial 5 = 120
$ escript factorial.zip 5
factorial 5 = 120
```

escript:create(FileOrBin, Sections) -> ok | {ok, binary()} | {error, term()}

Types:

```
FileOrBin = filename() | ’binary’
Sections = [Header] Body | Body
Header = shebang | {shebang, Shebang} | comment | {comment, Comment} | {emu_args, EmuArgs}
Shebang = string() | ’default’ | ’undefined’
Comment = string() | ’default’ | ’undefined’
EmuArgs = string() | ’undefined’
Body = {source, SourceCode} | {beam, BeamCode} | {archive, ZipArchive}
SourceCode = BeamCode = ZipArchive = binary()
```

The *create/2* function creates an escript from a list of sections. The sections can be given in any order. An escript begins with an optional *Header* followed by a mandatory *Body*. If the header is present, it does always begin with a *shebang*, possibly followed by a *comment* and *emu_args*. The *shebang* defaults to *"/usr/bin/env escript"*. The comment defaults to *"This is an -*- erlang -*- file"*. The created escript can either be returned as a binary or written to file.

As an example of how the function can be used, we create an interpreted escript which uses emu_args to set some emulator flag. In this case it happens to disable the smp_support. We do also extract the different sections from the newly created script:

```
> Source = "%% Demo\nmain(_Args) ->\n io:format(erlang:system_info(smp_support)).\n".
"%% Demo\nmain(_Args) ->\n    io:format(erlang:system_info(smp_support)).\n"
> io:format("~s\n", [Source]).
%% Demo
main(_Args) ->
    io:format(erlang:system_info(smp_support)).

ok
> {ok, Bin} = escript:create(binary, [shebang, comment, {emu_args, "-smp disable"}, {source, list_to_binary(So
{ok,<<"#!/usr/bin/env escript\n%% This is an -*- erlang -*- file\n%%!-smp disabl"...>>}
> file:write_file("demo.escript", Bin).
ok
> os:cmd("escript demo.escript").
"false"
> escript:extract("demo.escript", []).
{ok,[{shebang,default}, {comment,default}, {emu_args,"-smp disable"},
     {source,<<"%% Demo\nmain(_Args) ->\n    io:format(erlang:system_info(smp_su"...>>}]}
```

An escript without header can be created like this:

```
> file:write_file("demo.erl", ["%% demo.erl\n-module(demo).\n-export([main/1]).\n\n", Source]).
ok
> {ok, _, BeamCode} = compile:file("demo.erl", [binary, debug_info]).
{ok,demo,
   <<70,79,82,49,0,0,2,208,66,69,65,77,65,116,111,109,0,0,0,
     79,0,0,0,9,4,100,...>>}
> escript:create("demo.beam", [{beam, BeamCode}]).
ok
> escript:extract("demo.beam", []).
{ok,[{shebang,undefined}, {comment,undefined}, {emu_args,undefined},
    {beam,<<70,79,82,49,0,0,3,68,66,69,65,77,65,116,
           111,109,0,0,0,83,0,0,0,9,...>>}]}
> os:cmd("escript demo.beam").
"true"
```

Here we create an archive script containing both Erlang code as well as beam code. Then we iterate over all files in the archive and collect their contents and some info about them.

```
> {ok, SourceCode} = file:read_file("demo.erl").
{ok,<<"%% demo.erl\n-module(demo).\n-export([main/1]).\n\n%% Demo\nmain(_Arg"...>>}
> escript:create("demo.escript", [shebang, {archive, [{"demo.erl", SourceCode}, {"demo.beam", BeamCode}], [
ok
> {ok, [{shebang,default}, {comment,undefined}, {emu_args,undefined}, {archive, ArchiveBin}]} = escript:extr
{ok,[{shebang,default}, {comment,undefined}, {emu_args,undefined},
    {{archive,<<80,75,3,4,20,0,0,0,8,0,118,7,98,60,105,
             152,61,93,107,0,0,0,118,0,...>>}]}
> file:write_file("demo.zip", ArchiveBin).
ok
> zip:foldl(fun(N, I, B, A) -> [{N, I(), B()} | A] end, [], "demo.zip").
{ok,[{"demo.beam",
      {file_info,748,regular,read_write,
                 {{2010,3,2},{0,59,22}},
                 {{2010,3,2},{0,59,22}},
                 {{2010,3,2},{0,59,22}},
                 54,1,0,0,0,0,0},
      <<70,79,82,49,0,0,2,228,66,69,65,77,65,116,111,109,0,0,0,
        83,0,0,...>>},
     {"demo.erl",
      {file_info,118,regular,read_write,
                 {{2010,3,2},{0,59,22}},
                 {{2010,3,2},{0,59,22}},
                 {{2010,3,2},{0,59,22}},
                 54,1,0,0,0,0,0},
      <<"%% demo.erl\n-module(demo).\n-export([main/1]).\n\n%% Demo\nmain(_Arg"...>>}]}
```

escript:extract(File, Options) -> {ok, Sections} | {error, term()}

Types:

File = filename()
Options = [] | [compile_source]
Sections = Headers Body
Headers = {shebang, Shebang} {comment, Comment} {emu_args, EmuArgs}

Shebang = string() | ’default’ | ’undefined’
Comment = string() | ’default’ | ’undefined’
EmuArgs = string() | ’undefined’
Body = {source, SourceCode} | {source, BeamCode} | {beam, BeamCode} | {archive, ZipArchive}
SourceCode = BeamCode = ZipArchive = binary()

The *extract/2* function parses an escript and extracts its sections. This is the reverse of *create/2*.

All sections are returned even if they do not exist in the escript. If a particular section happens to have the same value as the default value, the extracted value is set to the atom *default*. If a section is missing, the extracted value is set to the atom *undefined*.

The *compile_source* option only affects the result if the escript contains *source* code. In that case the Erlang code is automatically compiled and *{source, BeamCode}* is returned instead of *{source, SourceCode}*.

```
> escript:create("demo.escript", [shebang, {archive, [{"demo.erl", SourceCode}, {"demo.beam", BeamCode}], [
ok
> {ok, [{shebang,default}, {comment,undefined}, {emu_args,undefined}, {archive, ArchiveBin}]} = escript:extr
{ok,[{{archive,<<80,75,3,4,20,0,0,0,8,0,118,7,98,60,105,
            152,61,93,107,0,0,0,118,0,...>>}
   {emu_args,undefined}]}
```

escript:script_name() -> File

Types:

File = filename()

The *script_name/0* function returns the name of the escript being executed. If the function is invoked outside the context of an escript, the behavior is undefined.

OPTIONS ACCEPTED BY ESCRIPT

-c:
Compile the escript regardless of the value of the mode attribute.

-d:
Debug the escript. Starts the debugger, loads the module containing the *main/1* function into the debugger, sets a breakpoint in *main/1* and invokes *main/1*. If the module is precompiled, it must be explicitly compiled with the *debug_info* option.

-i:
Interpret the escript regardless of the value of the mode attribute.

-s:
Only perform a syntactic and semantic check of the script file. Warnings and errors (if any) are written to the standard output, but the script will not be run. The exit status will be 0 if there were no errors, and 127 otherwise.

-n:
Compile the escript using the +native flag.

NAME

erl − The Erlang Emulator

DESCRIPTION

The *erl* program starts an Erlang runtime system. The exact details (for example, whether *erl* is a script or a program and which other programs it calls) are system-dependent.

Windows users probably wants to use the *werl* program instead, which runs in its own window with scrollbars and supports command-line editing. The *erl* program on Windows provides no line editing in its shell, and on Windows 95 there is no way to scroll back to text which has scrolled off the screen. The *erl* program must be used, however, in pipelines or if you want to redirect standard input or output.

Note:

As of ERTS version 5.8 (OTP-R14A) the runtime system will by default bind schedulers to logical processors using the *default_bind* bind type if the amount of schedulers are at least equal to the amount of logical processors configured, binding of schedulers is supported, and a CPU topology is available at startup.

If the Erlang runtime system is the only operating system process that binds threads to logical processors, this improves the performance of the runtime system. However, if other operating system processes (as for example another Erlang runtime system) also bind threads to logical processors, there might be a performance penalty instead. If this is the case you, are are advised to unbind the schedulers using the **+sbtu** command line argument, or by invoking **erlang:system_flag(scheduler_bind_type, unbound)**.

EXPORTS

erl <arguments>

Starts an Erlang runtime system.

The arguments can be divided into *emulator flags*, *flags* and *plain arguments*:

* Any argument starting with the character + is interpreted as an **emulator flag**.

  As indicated by the name, emulator flags controls the behavior of the emulator.

* Any argument starting with the character - (hyphen) is interpreted as a **flag** which should be passed to the Erlang part of the runtime system, more specifically to the *init* system process, see **init(3erl)**.

  The *init* process itself interprets some of these flags, the *init flags*. It also stores any remaining flags, the *user flags*. The latter can be retrieved by calling *init:get_argument/1*.

  It can be noted that there are a small number of "-" flags which now actually are emulator flags, see the description below.

* Plain arguments are not interpreted in any way. They are also stored by the *init* process and can be retrieved by calling *init:get_plain_arguments/0*. Plain arguments can occur before the first flag, or after a -- flag. Additionally, the flag *-extra* causes everything that follows to become plain arguments.

Example:

```
% erl +W w -sname arnie +R 9 -s my_init -extra +bertie
(arnie@host)1> init:get_argument(sname).
{ok,[["arnie"]]}
(arnie@host)2> init:get_plain_arguments().
["+bertie"]
```

Here *+W w* and *+R 9* are emulator flags. *-s my_init* is an init flag, interpreted by *init*. *-sname arnie* is a user flag, stored by *init*. It is read by Kernel and will cause the Erlang runtime system to become distributed. Finally, everything after *-extra* (that is, *+bertie*) is considered as plain arguments.

```
% erl -myflag 1
1> init:get_argument(myflag).
{ok,[["1"]]}
2> init:get_plain_arguments().
[]
```

Here the user flag *-myflag 1* is passed to and stored by the *init* process. It is a user defined flag, presumably used by some user defined application.

FLAGS

In the following list, init flags are marked (init flag). Unless otherwise specified, all other flags are user flags, for which the values can be retrieved by calling *init:get_argument/1*. Note that the list of user flags is not exhaustive, there may be additional, application specific flags which instead are documented in the corresponding application documentation.

--(init flag):
Everything following -- up to the next flag (*-flag* or *+flag*) is considered plain arguments and can be retrieved using *init:get_plain_arguments/0*.

-Application Par Val:
Sets the application configuration parameter *Par* to the value *Val* for the application *Application*, see **app(5)** and **application(3erl)**.

-args_file FileName:
Command line arguments are read from the file *FileName*. The arguments read from the file replace the '*-args_file FileName*' flag on the resulting command line.

The file *FileName* should be a plain text file and may contain comments and command line arguments. A comment begins with a # character and continues until next end of line character. Backslash (\\) is used as quoting character. All command line arguments accepted by *erl* are allowed, also the *-args_file FileName* flag. Be careful not to cause circular dependencies between files containing the *-args_file* flag, though.

The *-extra* flag is treated specially. Its scope ends at the end of the file. Arguments following an *-extra* flag are moved on the command line into the *-extra* section, i.e. the end of the command line following after an *-extra* flag.

-async_shell_start:
The initial Erlang shell does not read user input until the system boot procedure has been completed (Erlang 5.4 and later). This flag disables the start synchronization feature and lets the shell start in parallel with the rest of the system.

-boot File:
Specifies the name of the boot file, *File.boot*, which is used to start the system. See **init(3erl)**. Unless *File* contains an absolute path, the system searches for *File.boot* in the current and *$ROOT/bin* directories.

Defaults to *$ROOT/bin/start.boot*.

-boot_var Var Dir:
If the boot script contains a path variable *Var* other than *$ROOT*, this variable is expanded to *Dir*. Used when applications are installed in another directory than *$ROOT/lib*, see **systools:make_script/1,2**.

-code_path_cache:
Enables the code path cache of the code server, see **code(3erl)**.

-*compile Mod1 Mod2 ...*:
: Compiles the specified modules and then terminates (with non-zero exit code if the compilation of some file did not succeed). Implies -*noinput*. Not recommended - use **erlc** instead.

-*config Config*:
: Specifies the name of a configuration file, *Config.config*, which is used to configure applications. See **app(5)** and **application(3erl)**.

-*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(3erl)**.

-*cookie Cookie*:
: Obsolete flag without any effect and common misspelling for -*setcookie*. Use -*setcookie* instead.

-*detached*:
: Starts the Erlang runtime system detached from the system console. Useful for running daemons and backgrounds processes. Implies -*noinput*.

-*emu_args*:
: Useful for debugging. Prints out the actual arguments sent to the emulator.

-*env Variable Value*:
: Sets the host OS environment variable *Variable* to the value *Value* for the Erlang runtime system. Example:

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

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

-*eval Expr*(init flag):
: Makes *init* evaluate the expression *Expr*, see **init(3erl)**.

-*extra*(init flag):
: Everything following -*extra* is considered plain arguments and can be retrieved using *init:get_plain_arguments/0*.

-*heart*:
: Starts heart beat monitoring of the Erlang runtime system. See **heart(3erl)**.

-*hidden*:
: Starts the Erlang runtime system as a hidden node, if it is run as a distributed node. Hidden nodes always establish hidden connections to all other nodes except for nodes in the same global group. Hidden connections are not published on either of the connected nodes, i.e. neither of the connected nodes are part of the result from *nodes/0* on the other node. See also hidden global groups, **global_group(3erl)**.

-*hosts Hosts*:
: Specifies the IP addresses for the hosts on which Erlang boot servers are running, see **erl_boot_server(3erl)**. This flag is mandatory if the -*loader inet* flag is present.

The IP addresses must be given in the standard form (four decimal numbers separated by periods, for example *"150.236.20.74"*. Hosts names are not acceptable, but a broadcast address (preferably limited to the local network) is.

-*id Id*:
: Specifies the identity of the Erlang runtime system. If it is run as a distributed node, *Id* must be identical to the name supplied together with the -*sname* or -*name* flag.

-init_debug:
Makes *init* write some debug information while interpreting the boot script.

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

-loader Loader:
Specifies the method used by *erl_prim_loader* to load Erlang modules into the system. See **erl_prim_loader(3erl)**. Two *Loader* methods are supported, *efile* and *inet*. *efile* means use the local file system, this is the default. *inet* means use a boot server on another machine, and the *-id*, *-hosts* and *-setcookie* flags must be specified as well. If *Loader* is something else, the user supplied *Loader* port program is started.

-make:
Makes the Erlang runtime system invoke *make:all()* in the current working directory and then terminate. See **make(3erl)**. Implies *-noinput*.

-man Module:
Displays the manual page for the Erlang module *Module*. Only supported on Unix.

-mode interactive | embedded:
Indicates if the system should load code dynamically (*interactive*), or if all code should be loaded during system initialization (*embedded*), see **code(3erl)**. Defaults to *interactive*.

-name Name:
Makes the Erlang runtime system into a distributed node. This flag invokes all network servers necessary for a node to become distributed. See **net_kernel(3erl)**. It is also ensured that *epmd* runs on the current host before Erlang is started. See **epmd(1)**.

The name of the node will be *Name@Host*, where *Host* is the fully qualified host name of the current host. For short names, use the *-sname* flag instead.

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

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

-nostick:
Disables the sticky directory facility of the Erlang code server, see **code(3erl)**.

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

-pa Dir1 Dir2 ...:
Adds the specified directories to the beginning of the code path, similar to *code:add_pathsa/1*. See **code(3erl)**. As an alternative to *-pa*, if several directories are to be prepended to the code and the directories have a common parent directory, that parent directory could be specified in the *ERL_LIBS* environment variable. See **code(3erl)**.

-pz Dir1 Dir2 ...:
Adds the specified directories to the end of the code path, similar to *code:add_pathsz/1*. See **code(3erl)**.

-remsh Node:
Starts Erlang with a remote shell connected to *Node*.

-rsh Program:
: Specifies an alternative to *rsh* for starting a slave node on a remote host. See **slave(3erl)**.

-run Mod [Func [Arg1, Arg2, ...]](init flag):
: Makes *init* call the specified function. *Func* defaults to *start*. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list *[Arg1,Arg2,...]* as argument. All arguments are passed as strings. See **init(3erl)**.

-s Mod [Func [Arg1, Arg2, ...]](init flag):
: Makes *init* call the specified function. *Func* defaults to *start*. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list *[Arg1,Arg2,...]* as argument. All arguments are passed as atoms. See **init(3erl)**.

-setcookie Cookie:
: Sets the magic cookie of the node to *Cookie*, see **erlang:set_cookie/2**.

-shutdown_time Time:
: Specifies how long time (in milliseconds) the *init* process is allowed to spend shutting down the system. If *Time* ms have elapsed, all processes still existing are killed. Defaults to *infinity*.

-sname Name:
: Makes the Erlang runtime system into a distributed node, similar to *-name*, but the host name portion of the node name *Name@Host* will be the short name, not fully qualified.

This is sometimes the only way to run distributed Erlang if the DNS (Domain Name System) is not running. There can be no communication between nodes running with the *-sname* flag and those running with the *-name* flag, as node names must be unique in distributed Erlang systems.

-smp [enable|auto|disable]:
: *-smp enable* and *-smp* starts the Erlang runtime system with SMP support enabled. This may fail if no runtime system with SMP support is available. *-smp auto* starts the Erlang runtime system with SMP support enabled if it is available and more than one logical processor are detected. *-smp disable* starts a runtime system without SMP support. By default *-smp auto* will be used unless a conflicting parameter has been passed, then *-smp disable* will be used. Currently only the *-hybrid* parameter conflicts with *-smp auto*.

NOTE: The runtime system with SMP support will not be available on all supported platforms. See also the **+S** flag.

-version(emulator flag):
: Makes the emulator print out its version number. The same as *erl +V*.

EMULATOR FLAGS

erl invokes the code for the Erlang emulator (virtual machine), which supports the following flags:

+a size:
: Suggested stack size, in kilowords, for threads in the async-thread pool. Valid range is 16-8192 kilowords. The default suggested stack size is 16 kilowords, i.e, 64 kilobyte on 32-bit architectures. This small default size has been chosen since the amount of async-threads might be quite large. The default size is enough for drivers delivered with Erlang/OTP, but might not be sufficiently large for other dynamically linked in drivers that use the **driver_async()** functionality. Note that the value passed is only a suggestion, and it might even be ignored on some platforms.

+A size:
: Sets the number of threads in async thread pool, valid range is 0-1024. Default is 0.

+B [c | d | i]:
: The *c* option makes *Ctrl-C* interrupt the current shell instead of invoking the emulator break handler. The *d* option (same as specifying *+B* without an extra option) disables the break handler. The *i* option makes the emulator ignore any break signal.

If the *c* option is used with *oldshell* on Unix, *Ctrl-C* will restart the shell process rather than interrupt it.

Note that on Windows, this flag is only applicable for *werl*, not *erl* (*oldshell*). Note also that *Ctrl-Break* is used instead of *Ctrl-C* on Windows.

+*c*:
Disable compensation for sudden changes of system time.

Normally, *erlang:now/0* will not immediately reflect sudden changes in the system time, in order to keep timers (including *receive-after*) working. Instead, the time maintained by *erlang:now/0* is slowly adjusted towards the new system time. (Slowly means in one percent adjustments; if the time is off by one minute, the time will be adjusted in 100 minutes.)

When the *+c* option is given, this slow adjustment will not take place. Instead *erlang:now/0* will always reflect the current system time. Note that timers are based on *erlang:now/0*. If the system time jumps, timers then time out at the wrong time.

+*d*:
If the emulator detects an internal error (or runs out of memory), it will by default generate both a crash dump and a core dump. The core dump will, however, not be very useful since the content of process heaps is destroyed by the crash dump generation.

The *+d* option instructs the emulator to only produce a core dump and no crash dump if an internal error is detected.

Calling *erlang:halt/1* with a string argument will still produce a crash dump.

+*e Number*:
Set max number of ETS tables.

+*ec*:
Force the *compressed* option on all ETS tables. Only intended for test and evaluation.

+*fnl*:
The VM works with file names as if they are encoded using the ISO-latin-1 encoding, disallowing Unicode characters with codepoints beyond 255. This is default on operating systems that have transparent file naming, i.e. all Unixes except MacOSX.

+*fnu*:
The VM works with file names as if they are encoded using UTF-8 (or some other system specific Unicode encoding). This is the default on operating systems that enforce Unicode encoding, i.e. Windows and MacOSX.

By enabling Unicode file name translation on systems where this is not default, you open up to the possibility that some file names can not be interpreted by the VM and therefore will be returned to the program as raw binaries. The option is therefore considered experimental.

+*fna*:
Selection between *+fnl* and *+fnu* is done based on the current locale settings in the OS, meaning that if you have set your terminal for UTF-8 encoding, the filesystem is expected to use the same encoding for filenames (use with care).

+*hms Size*:
Sets the default heap size of processes to the size *Size*.

+*hmbs Size*:
Sets the default binary virtual heap size of processes to the size *Size*.

+*K true | false*:
Enables or disables the kernel poll functionality if the emulator supports it. Default is *false* (disabled). If the emulator does not support kernel poll, and the *+K* flag is passed to the emulator, a warning is issued at startup.

+*l*:
Enables auto load tracing, displaying info while loading code.

+*MFlag Value*:
Memory allocator specific flags, see **erts_alloc(3erl)** for further information.

+*P Number*:
Sets the maximum number of concurrent processes for this system. *Number* must be in the range 16..134217727. Default is 32768.

+*R ReleaseNumber*:
Sets the compatibility mode.

The distribution mechanism is not backwards compatible by default. This flags sets the emulator in compatibility mode with an earlier Erlang/OTP release *ReleaseNumber*. The release number must be in the range *7..<current release>*. This limits the emulator, making it possible for it to communicate with Erlang nodes (as well as C- and Java nodes) running that earlier release.

For example, an R10 node is not automatically compatible with an R9 node, but R10 nodes started with the *+R 9* flag can co-exist with R9 nodes in the same distributed Erlang system, they are R9-compatible.

Note: Make sure all nodes (Erlang-, C-, and Java nodes) of a distributed Erlang system is of the same Erlang/OTP release, or from two different Erlang/OTP releases X and Y, where *all* Y nodes have compatibility mode X.

For example: A distributed Erlang system can consist of R10 nodes, or of R9 nodes and R9-compatible R10 nodes, but not of R9 nodes, R9-compatible R10 nodes and "regular" R10 nodes, as R9 and "regular" R10 nodes are not compatible.

+*r*:
Force ets memory block to be moved on realloc.

+*rg ReaderGroupsLimit*:
Limits the amount of reader groups used by read/write locks optimized for read operations in the Erlang runtime system. By default the reader groups limit equals 8.

When the amount of schedulers is less than or equal to the reader groups limit, each scheduler has its own reader group. When the amount of schedulers is larger than the reader groups limit, schedulers share reader groups. Shared reader groups degrades read lock and read unlock performance while a large amount of reader groups degrades write lock performance, so the limit is a tradeoff between performance for read operations and performance for write operations. Each reader group currently consumes 64 byte in each read/write lock. Also note that a runtime system using shared reader groups benefits from **binding schedulers to logical processors**, since the reader groups are distributed better between schedulers.

+*S Schedulers:SchedulerOnline*:
Sets the amount of scheduler threads to create and scheduler threads to set online when SMP support has been enabled. Valid range for both values are 1-1024. If the Erlang runtime system is able to

determine the amount of logical processors configured and logical processors available, *Schedulers* will default to logical processors configured, and *SchedulersOnline* will default to logical processors available; otherwise, the default values will be 1. *Schedulers* may be omitted if *:SchedulerOnline* is not and vice versa. The amount of schedulers online can be changed at run time via **erlang:system_flag(schedulers_online, SchedulersOnline)**.

This flag will be ignored if the emulator doesn't have SMP support enabled (see the **-smp** flag).

+sFlag Value:
Scheduling specific flags.

+sbt BindType:
Set scheduler bind type. Currently valid *BindType*s:

u:
Same as **erlang:system_flag(scheduler_bind_type, unbound)**.

ns:
Same as **erlang:system_flag(scheduler_bind_type, no_spread)**.

ts:
Same as **erlang:system_flag(scheduler_bind_type, thread_spread)**.

ps:
Same as **erlang:system_flag(scheduler_bind_type, processor_spread)**.

s:
Same as **erlang:system_flag(scheduler_bind_type, spread)**.

nnts:
Same as **erlang:system_flag(scheduler_bind_type, no_node_thread_spread)**.

nnps:
Same as **erlang:system_flag(scheduler_bind_type, no_node_processor_spread)**.

tnnps:
Same as **erlang:system_flag(scheduler_bind_type, thread_no_node_processor_spread)**.

db:
Same as **erlang:system_flag(scheduler_bind_type, default_bind)**.

Binding of schedulers is currently only supported on newer Linux, Solaris, FreeBSD, and Windows systems.

If no CPU topology is available when the *+sbt* flag is processed and *BindType* is any other type than *u*, the runtime system will fail to start. CPU topology can be defined using the **+sct** flag. Note that the *+sct* flag may have to be passed before the *+sbt* flag on the command line (in case no CPU topology has been automatically detected).

The runtime system will by default bind schedulers to logical processors using the *default_bind* bind type if the amount of schedulers are at least equal to the amount of logical processors configured, binding of schedulers is supported, and a CPU topology is available at startup.

NOTE: If the Erlang runtime system is the only operating system process that binds threads to logical processors, this improves the performance of the runtime system. However, if other operating system processes (as for example another Erlang runtime system) also bind threads to logical processors, there might be a performance penalty instead. If this is the case you, are advised to unbind the schedulers using the *+sbtu* command line argument, or by invoking **erlang:system_flag(scheduler_bind_type, unbound)**.

For more information, see **erlang:system_flag(scheduler_bind_type, SchedulerBindType)**.

+sct CpuTopology:

* *<Id> = integer(); when 0 =< <Id> =< 65535*
* *<IdRange> = <Id>-<Id>*
* *<IdOrIdRange> = <Id> | <IdRange>*
* *<IdList> = <IdOrIdRange>,<IdOrIdRange> | <IdOrIdRange>*
* *<LogicalIds> = L<IdList>*
* *<ThreadIds> = T<IdList> | t<IdList>*
* *<CoreIds> = C<IdList> | c<IdList>*
* *<ProcessorIds> = P<IdList> | p<IdList>*
* *<NodeIds> = N<IdList> | n<IdList>*
* *<IdDefs> = <LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds> | <LogicalIds><ThreadIds><CoreIds><NodeIds><ProcessorIds>*
* *CpuTopology = <IdDefs>:<IdDefs> | <IdDefs>*

Upper-case letters signify real identifiers and lower-case letters signify fake identifiers only used for description of the topology. Identifiers passed as real identifiers may be used by the runtime system when trying to access specific hardware and if they are not correct the behavior is undefined. Faked logical CPU identifiers are not accepted since there is no point in defining the CPU topology without real logical CPU identifiers. Thread, core, processor, and node identifiers may be left out. If left out, thread id defaults to *t0*, core id defaults to *c0*, processor id defaults to *p0*, and node id will be left undefined. Either each logical processor must belong to one and only one NUMA node, or no logical processors must belong to any NUMA nodes.

Both increasing and decreasing *<IdRange>*s are allowed.

NUMA node identifiers are system wide. That is, each NUMA node on the system have to have a unique identifier. Processor identifiers are also system wide. Core identifiers are processor wide. Thread identifiers are core wide.

The order of the identifier types imply the hierarchy of the CPU topology. Valid orders are either *<LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds>*, or *<LogicalIds><ThreadIds><CoreIds><NodeIds><ProcessorIds>*. That is, thread is part of a core which is part of a processor which is part of a NUMA node, or thread is part of a core which is part of a NUMA node which is part of a processor. A cpu topology can consist of both processor external, and processor internal NUMA nodes as long as each logical processor belongs to one and only one NUMA node. If *<ProcessorIds>* is left out, its default position will be before *<NodeIds>*. That is, the default is processor external NUMA nodes.

If a list of identifiers is used in an *<IdDefs>*:

* *<LogicalIds>* have to be a list of identifiers.
* At least one other identifier type apart from *<LogicalIds>* also have to have a list of identifiers.
* All lists of identifiers have to produce the same amount of identifiers.

A simple example. A single quad core processor may be described this way:

```
% erl +sct L0-3c0-3
1> erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
            {core,{logical,1}},
            {core,{logical,2}},
            {core,{logical,3}}]}]
```

A little more complicated example. Two quad core processors. Each processor in its own NUMA node. The ordering of logical processors is a little weird. This in order to give a better example of identifier lists:

```
% erl +sct L0-1,3-2c0-3p0N0:L7,4,6-5c0-3p1N1
1> erlang:system_info(cpu_topology).
[{node,[{processor,[{core,{logical,0}},
                    {core,{logical,1}},
                    {core,{logical,3}},
                    {core,{logical,2}}]}]},
 {node,[{processor,[{core,{logical,7}},
                    {core,{logical,4}},
                    {core,{logical,6}},
                    {core,{logical,5}}]}]}]
```

As long as real identifiers are correct it is okay to pass a CPU topology that is not a correct description of the CPU topology. When used with care this can actually be very useful. This in order to trick the emulator to bind its schedulers as you want. For example, if you want to run multiple Erlang runtime systems on the same machine, you want to reduce the amount of schedulers used and manipulate the CPU topology so that they bind to different logical CPUs. An example, with two Erlang runtime systems on a quad core machine:

```
% erl +sct L0-3c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname one
% erl +sct L3-0c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname two
```

In this example each runtime system have two schedulers each online, and all schedulers online will run on different cores. If we change to one scheduler online on one runtime system, and three schedulers online on the other, all schedulers online will still run on different cores.

Note that a faked CPU topology that does not reflect how the real CPU topology looks like is likely to decrease the performance of the runtime system.

For more information, see **erlang:system_flag(cpu_topology, CpuTopology)**.

+swt very_low|low|medium|high|very_high:
Set scheduler wakeup threshold. Default is *medium*. The threshold determines when to wake up sleeping schedulers when more work than can be handled by currently awake schedulers exist. A low threshold will cause earlier wakeups, and a high threshold will cause later wakeups. Early wakeups will distribute work over multiple schedulers faster, but work will more easily bounce between schedulers.

NOTE: This flag may be removed or changed at any time without prior notice.

+sss size:

Suggested stack size, in kilowords, for scheduler threads. Valid range is 4-8192 kilowords. The default stack size is OS dependent.

+t size:

Set the maximum number of atoms the VM can handle. Default is 1048576.

+T Level:

Enables modified timing and sets the modified timing level. Currently valid range is 0-9. The timing of the runtime system will change. A high level usually means a greater change than a low level. Changing the timing can be very useful for finding timing related bugs.

Currently, modified timing affects the following:

Process spawning:

A process calling *spawn*, *spawn_link*, *spawn_monitor*, or *spawn_opt* will be scheduled out immediately after completing the call. When higher modified timing levels are used, the caller will also sleep for a while after being scheduled out.

Context reductions:

The amount of reductions a process is a allowed to use before being scheduled out is increased or reduced.

Input reductions:

The amount of reductions performed before checking I/O is increased or reduced.

NOTE: Performance will suffer when modified timing is enabled. This flag is *only* intended for testing and debugging. Also note that *return_to* and *return_from* trace messages will be lost when tracing on the spawn BIFs. This flag may be removed or changed at any time without prior notice.

+V:

Makes the emulator print out its version number.

+v:

Verbose.

+W w | i:

Sets the mapping of warning messages for *error_logger*. Messages sent to the error logger using one of the warning routines can be mapped either to errors (default), warnings (*+W w*), or info reports (*+W i*). The current mapping can be retrieved using *error_logger:warning_map/0*. See **error_logger(3erl)** for further information.

+zFlag Value:

Miscellaneous flags.

+zdbbl size:

Set the distribution buffer busy limit (**dist_buf_busy_limit**) in kilobytes. Valid range is 1-2097151. Default is 1024.

A larger buffer limit will allow processes to buffer more outgoing messages over the distribution. When the buffer limit has been reached, sending processes will be suspended until the buffer size has shrunk. The buffer limit is per distribution channel. A higher limit will give lower latency and higher throughput at the expense of higher memory usage.

ENVIRONMENT VARIABLES

ERL_CRASH_DUMP:

If the emulator needs to write a crash dump, the value of this variable will be the file name of the crash dump file. If the variable is not set, the name of the crash dump file will be *erl_crash.dump* in the current directory.

ERL_CRASH_DUMP_NICE:

Unix systems: If the emulator needs to write a crash dump, it will use the value of this variable to set the nice value for the process, thus lowering its priority. The allowable range is 1 through 39 (higher values will be replaced with 39). The highest value, 39, will give the process the lowest priority.

ERL_CRASH_DUMP_SECONDS:

Unix systems: This variable gives the number of seconds that the emulator will be allowed to spend writing a crash dump. When the given number of seconds have elapsed, the emulator will be terminated by a SIGALRM signal.

ERL_AFLAGS:

The content of this environment variable will be added to the beginning of the command line for *erl*.

The *-extra* flag is treated specially. Its scope ends at the end of the environment variable content. Arguments following an *-extra* flag are moved on the command line into the *-extra* section, i.e. the end of the command line following after an *-extra* flag.

*ERL_ZFLAGS*and *ERL_FLAGS*:

The content of these environment variables will be added to the end of the command line for *erl*.

The *-extra* flag is treated specially. Its scope ends at the end of the environment variable content. Arguments following an *-extra* flag are moved on the command line into the *-extra* section, i.e. the end of the command line following after an *-extra* flag.

ERL_LIBS:

This environment variable contains a list of additional library directories that the code server will search for applications and add to the code path. See **code(3erl)**.

ERL_EPMD_ADDRESS:

This environment variable may be set to a comma-separated list of IP addresses, in which case the **epmd** daemon will listen only on the specified address(es) and on the loopback address (which is implicitly added to the list if it has not been specified).

ERL_EPMD_PORT:

This environment variable can contain the port number to use when communicating with **epmd**. The default port will work fine in most cases. A different port can be specified to allow nodes of independent clusters to co-exist on the same host. All nodes in a cluster must use the same epmd port number.

CONFIGURATION

The standard Erlang/OTP system can be re-configured to change the default behavior on start-up.

The .erlang Start-up File:

When Erlang/OTP is started, the system searches for a file named .erlang in the directory where Erlang/OTP is started. If not found, the user’s home directory is searched for an .erlang file.

If an .erlang file is found, it is assumed to contain valid Erlang expressions. These expressions are evaluated as if they were input to the shell.

A typical .erlang file contains a set of search paths, for example:

```
io:format("executing user profile in HOME/.erlang\n",[]).
code:add_path("/home/calvin/test/ebin").
code:add_path("/home/hobbes/bigappl-1.2/ebin").
io:format(".erlang rc finished\n",[]).
```

user_default and shell_default:

Functions in the shell which are not prefixed by a module name are assumed to be functional objects (Funs), built-in functions (BIFs), or belong to the module user_default or shell_default.

To include private shell commands, define them in a module user_default and add the following argument as the first line in the .erlang file.

```
code:load_abs("..../user_default").
```

erl:

If the contents of .erlang are changed and a private version of user_default is defined, it is possible to customize the Erlang/OTP environment. More powerful changes can be made by supplying command line arguments in the start-up script erl. Refer to erl(1) and **init(3erl)** for further information.

SEE ALSO

init(3erl), **erl_prim_loader(3erl)**, **erl_boot_server(3erl)**, **code(3erl)**, **application(3erl)**, **heart(3erl)**, **net_kernel(3erl)**, **auth(3erl)**, **make(3erl)**, **epmd(1)**, **erts_alloc(3erl)**

NAME

erl_call − Call/Start a Distributed Erlang Node

DESCRIPTION

erl_call makes it possible to start and/or communicate with a distributed Erlang node. It is built upon the *erl_interface* library as an example application. Its purpose is to use an Unix shell script to interact with a distributed Erlang node. It performs all communication with the Erlang *rex server,* using the standard Erlang RPC facility. It does not require any special software to be run at the Erlang target node.

The main use is to either start a distributed Erlang node or to make an ordinary function call. However, it is also possible to pipe an Erlang module to *erl_call* and have it compiled, or to pipe a sequence of Erlang expressions to be evaluated (similar to the Erlang shell).

Options, which cause *stdin* to be read, can be used with advantage as scripts from within (Unix) shell scripts. Another nice use of *erl_call* could be from (http) CGI-bin scripts.

EXPORTS

erl_call <options>

Each option flag is described below with its name, type and meaning.

-a [Mod [Fun [Args]]]]:
(*optional*): Applies the specified function and returns the result. *Mod* must be specified, however *start* and *[]* are assumed for unspecified *Fun* and *Args*, respectively. *Args* should be in the same format as for *erlang:apply/3*. Note that this flag takes exactly one argument, so quoting may be necessary in order to group *Mod*, *Fun* and *Args*, in a manner dependent on the behavior of your command shell.

-c Cookie:
(*optional*): Use this option to specify a certain cookie. If no cookie is specified, the *˜/.erlang.cookie* file is read and its content are used as cookie. The Erlang node we want to communicate with must have the same cookie.

-d:
(*optional*): Debug mode. This causes all IO to be output to the file *˜/.erl_call.out.Nodename*, where *Nodename* is the node name of the Erlang node in question.

-e:
(*optional*): Reads a sequence of Erlang expressions, separated by ’,’ and ended with a ’.’, from *stdin* until EOF (Control-D). Evaluates the expressions and returns the result from the last expression. Returns *{ok,Result}* if successful.

-h HiddenName:
(*optional*): Specifies the name of the hidden node that *erl_call* represents.

-m:
(*optional*): Reads an Erlang module from *stdin* and compiles it.

-n Node:
(one of *-n, -name, -sname* is required): Has the same meaning as *-name* and can still be used for backwards compatibility reasons.

-name Node:
(one of *-n, -name, -sname* is required): *Node* is the name of the node to be started or communicated with. It is assumed that *Node* is started with *erl -name*, which means that fully qualified long node names are used. If the *-s* option is given, an Erlang node will (if necessary) be started with *erl -name.*

-q:
(*optional*): Halts the Erlang node specified with the -n switch. This switch overrides the -s switch.

-r:
(*optional*): Generates a random name of the hidden node that *erl_call* represents.

-s:
(*optional*): Starts a distributed Erlang node if necessary. This means that in a sequence of calls, where the '*-s*' and '*-n Node*' are constant, only the first call will start the Erlang node. This makes the rest of the communication very fast. This flag is currently only available on the Unix platform.

-sname Node:
(one of *-n, -name, -sname* is required): *Node* is the name of the node to be started or communicated with. It is assumed that *Node* is started with *erl -sname* which means that short node names are used. If *-s* option is given, an Erlang node will be started (if necessary) with *erl -sname*.

-v:
(*optional*): Prints a lot of *verbose* information. This is only useful for the developer and maintainer of *erl_call*.

-x ErlScript:
(*optional*): Specifies another name of the Erlang start-up script to be used. If not specified, the standard *erl* start-up script is used.

EXAMPLES

Starts an Erlang node and calls *erlang:time/0*.

```
erl_call -s -a 'erlang time' -n madonna
{18,27,34}
```

Terminates an Erlang node by calling *erlang:halt/0*.

```
erl_call -s -a 'erlang halt' -n madonna
```

An apply with several arguments.

```
erl_call -s -a 'lists map [{math,sqrt},[1,4,9,16,25]]' -n madonna
```

Evaluates a couple of expressions. **The input ends with EOF (Control-D)** .

```
erl_call -s -e -n madonna
statistics(runtime),
X=1,
Y=2,
{_,T}=statistics(runtime),
{X+Y,T}.
^D
{ok,{3,0}}
```

Compiles a module and runs it. **Again, the input ends with EOF (Control-D)** . (In the example shown, the output has been formatted afterwards).

```
erl_call -s -m -a lolita -n madonna
-module(lolita).
-compile(export_all).
start() ->
    P = processes(),
    F = fun(X) -> {X,process_info(X,registered_name)} end,
    lists:map(F,[],P).
```

```
ˆD
[{<madonna@chivas.du.etx.ericsson.se,0,0>,
              {registered_name,init}},
 {<madonna@chivas.du.etx.ericsson.se,2,0>,
              {registered_name,erl_prim_loader}},
 {<madonna@chivas.du.etx.ericsson.se,4,0>,
              {registered_name,error_logger}},
 {<madonna@chivas.du.etx.ericsson.se,5,0>,
              {registered_name,application_controller}},
 {<madonna@chivas.du.etx.ericsson.se,6,0>,
              {registered_name,kernel}},
 {<madonna@chivas.du.etx.ericsson.se,7,0>,
              []},
 {<madonna@chivas.du.etx.ericsson.se,8,0>,
              {registered_name,kernel_sup}},
 {<madonna@chivas.du.etx.ericsson.se,9,0>,
              {registered_name,net_sup}},
 {<madonna@chivas.du.etx.ericsson.se,10,0>,
              {registered_name,net_kernel}},
 {<madonna@chivas.du.etx.ericsson.se,11,0>,
              []},
 {<madonna@chivas.du.etx.ericsson.se,12,0>,
              {registered_name,global_name_server}},
 {<madonna@chivas.du.etx.ericsson.se,13,0>,
              {registered_name,auth}},
 {<madonna@chivas.du.etx.ericsson.se,14,0>,
              {registered_name,rex}},
 {<madonna@chivas.du.etx.ericsson.se,15,0>,
              []},
 {<madonna@chivas.du.etx.ericsson.se,16,0>,
              {registered_name,file_server}},
 {<madonna@chivas.du.etx.ericsson.se,17,0>,
              {registered_name,code_server}},
 {<madonna@chivas.du.etx.ericsson.se,20,0>,
              {registered_name,user}},
 {<madonna@chivas.du.etx.ericsson.se,38,0>,
              []}]
```