NAME

c - Command interface module.

DESCRIPTION

This module enables users to enter the short form of some commonly used commands.

Note:

These functions are intended for interactive use in the Erlang shell only. The module prefix can be omitted.

EXPORTS

bt(Pid) -> ok | undefined

Types:

Pid = pid()

Stack backtrace for a process. Equivalent to *erlang:process_display(Pid, backtrace)*.

c(Module) -> {ok, ModuleName} | error

Types:

Module = file:name()
ModuleName = module()

Works like *c(Module, [])*.

c(Module, Options) -> {ok, ModuleName} | error

Types:

Module = file:name()
Options = [compile:option()] | compile:option()
ModuleName = module()

Compiles and then purges and loads the code for a module. *Module* can be either a module name

or a source file path, with or without *.erl* extension.

If *Module* is a string, it is assumed to be a source file path, and the compiler will attempt to compile the source file with the options *Options*. If compilation fails, the old object file (if any) is deleted.

If *Module* is an atom, a source file with that exact name or with *.erl* extension will be looked for. If found, the source file is compiled with the options *Options*. If compilation fails, the old object file (if any) is deleted.

If *Module* is an atom and is not the path of a source file, then the code path is searched to locate the object file for the module and extract its original compiler options and source path. If the source file is not found in the original location, *filelib:find_source/1* is used to search for it relative to the directory of the object file.

The source file is compiled with the the original options appended to the given *Options*, the output replacing the old object file if and only if compilation succeeds.

Notice that purging the code means that any processes lingering in old code for the module are killed without warning. For more information, see the code module.

c(Module, Options, Filter) -> {ok, ModuleName} | error

Types:

```
Module = atom()
Options = [compile:option()]
Filter = fun((compile:option()) -> boolean())
ModuleName = module()
```

Compiles and then purges and loads the code for module *Module*, which must be an atom.

The code path is searched to locate the object file for module *Module* and extract its original compiler options and source path. If the source file is not found in the original location, *filelib:find_source/1* is used to search for it relative to the directory of the object file.

The source file is compiled with the the original options appended to the given *Options*, the output replacing the old object file if and only if compilation succeeds. The function *Filter* specifies which elements to remove from the original compiler options before the new options are added. The

Filter fun should return *true* for options to keep, and *false* for options to remove.

Notice that purging the code means that any processes lingering in old code for the module are killed without warning. For more information, see the code module.

cd(Dir) -> ok

Types:

Dir = file:name()

Changes working directory to *Dir*, which can be a relative name, and then prints the name of the new working directory.

Example:

```
2> cd("../erlang").
/home/ron/erlang
```

erlangrc(PathList) -> {ok, file:filename()} | {error, term()}

Types:

PathList = [Dir :: file:name()]

Search *PathList* and load *.erlang* resource file if found.

flush() -> ok

Flushes any messages sent to the shell.

help() -> ok

Displays help information: all valid shell internal commands, and commands in this module.

h(Module :: module()) -> h_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}

Print the documentation for *Module*

h(Module :: module(), Function :: function()) -> hf_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
hf_return() = h_return() | {error, function_missing}

Print the documentation for all *Module:Function*s (regardless of arity).

h(Module :: module(), Function :: function(), Arity :: arity()) ->
hf_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
hf_return() = h_return() | {error, function_missing}

Print the documentation for *Module:Function/Arity*.

hcb(Module :: module()) -> h_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}

Print the callback documentation for *Module*

hcb(Module :: module(), Callback :: atom()) -> hcb_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
hcb_return() = h_return() | {error, callback_missing}

Print the callback documentation for all *Module:Callback*s (regardless of arity).

hcb(Module :: module(), Callback :: atom(), Arity :: arity()) ->
hcb_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
hcb_return() = h_return() | {error, callback_missing}

Print the callback documentation for *Module:Callback/Arity*.

ht(Module :: module()) -> h_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}

Print the type documentation for *Module*

ht(Module :: module(), Type :: atom()) -> ht_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
ht_return() = h_return() | {error, type_missing}

Print the type documentation for *Type* in *Module* regardless of arity.

ht(Module :: module(), Type :: atom(), Arity :: arity()) ->
ht_return()

Types:

h_return() =
ok | {error, missing | {unknown_format, unicode:chardata()}}
ht_return() = h_return() | {error, type_missing}

Print the type documentation for *Type/Arity* in *Module*.

i() -> ok

ni() -> ok

i/0 displays system information, listing information about all processes. *ni/0* does the same, but for all nodes the network.

i(X, Y, Z) -> [{atom(), term()}]

Types:

X = Y = Z = integer() >= 0

Displays information about a process, Equivalent to *process_info(pid(X, Y, Z))*, but location transparent.

l(Module) -> code:load_ret()

Types:

Module = module()

Purges and loads, or reloads, a module by calling *code:purge(Module)* followed by *code:load_file(Module)*.

Notice that purging the code means that any processes lingering in old code for the module are killed without warning. For more information, see *code/3*.

lc(Files) -> ok

Types:

Files = [File]
File

Compiles a list of files by calling *compile:file(File, [report_errors, report_warnings])* for each *File* in *Files*.

For information about *File*, see *file:filename()*.

lm() -> [code:load_ret()]

Reloads all currently loaded modules that have changed on disk (see *mm()*). Returns the list of results from calling *l(M)* for each such *M*.

ls() -> ok

Lists files in the current directory.

ls(Dir) -> ok

Types:

Dir = file:name()

Lists files in directory *Dir* or, if *Dir* is a file, only lists it.

m() -> ok

Displays information about the loaded modules, including the files from which they have been loaded.

m(Module) -> ok

Types:

Module = module()

Displays information about *Module*.

mm() -> [module()]

Lists all modified modules. Shorthand for *code:modified_modules/0*.

memory() -> [{Type, Size}]

Types:

Type = atom()
Size = integer() >= 0

Memory allocation information. Equivalent to *erlang:memory/0*.

memory(Type) -> Size

memory(Types) -> [{Type, Size}]

Types:

Types = [Type]
Type = atom()
Size = integer() >= 0

Memory allocation information. Equivalent to *erlang:memory/1*.

nc(File) -> {ok, Module} | error

nc(File, Options) -> {ok, Module} | error

Types:

File = file:name()
Options = [Option] | Option
Option = compile:option()
Module = module()

Compiles and then loads the code for a file on all nodes. *Options* defaults to *[]*. Compilation is equivalent to:

compile:file(File, Options ++ [report_errors, report_warnings])

nl(Module) -> abcast | error

Types:

Module = module()

Loads *Module* on all nodes.

pid(X, Y, Z) -> pid()

Types:

X = Y = Z = integer() >= 0

Converts *X*, *Y*, *Z* to pid <*X.Y.Z*>. This function is only to be used when debugging.

pwd() -> ok

Prints the name of the working directory.

q() -> no_return()

This function is shorthand for *init:stop()*, that is, it causes the node to stop in a controlled fashion.

regs() -> ok

nregs() -> ok

regs/0 displays information about all registered processes. *nregs/0* does the same, but for all nodes in the network.

uptime() -> ok

Prints the node uptime (as specified by *erlang:statistics(wall_clock)*) in human-readable form.

xm(ModSpec) -> void()

Types:

```
ModSpec = Module | Filename
 Module = atom()
 Filename = string()
```

Finds undefined functions, unused functions, and calls to deprecated functions in a module by calling *xref:m/1*.

y(File) -> YeccRet

Types:

```
File = name()
YeccRet
```

Generates an LALR-1 parser. Equivalent to:

yecc:file(File)

For information about *File = name()*, see *filename(3)*. For information about *YeccRet*, see *yecc:file/2*.

y(File, Options) -> YeccRet

Types:

File = name()
Options, YeccRet

Generates an LALR-1 parser. Equivalent to:

yecc:file(File, Options)

For information about *File = name()*, see *filename(3)*. For information about *Options* and *YeccRet*, see *yecc:file/2*.

SEE ALSO

filename(3), *compile(3)*, *erlang(3)*, *yecc(3)*, *xref(3)*