NAME

int - Interpreter Interface.

DESCRIPTION

The Erlang interpreter provides mechanisms for breakpoints and stepwise execution of code. It is primarily intended to be used by Debugger, see the User's Guide and *debugger(3)*.

The following can be done from the shell:

* Specify the modules to be interpreted.

* Specify breakpoints.

* Monitor the current status of all processes executing code in interpreted modules, also processes at other Erlang nodes.

By *attaching to* a process executing interpreted code, it is possible to examine variable bindings and order stepwise execution. This is done by sending and receiving information to/from the process through a third process, called the meta process. You can implement your own attached process. See *int.erl* for available functions and *dbg_wx_trace.erl* for possible messages.

The interpreter depends on the Kernel, STDLIB, and GS applications. This means that modules belonging to any of these applications are not allowed to be interpreted, as it could lead to a deadlock or emulator crash. This also applies to modules belonging to the Debugger application.

BREAKPOINTS

Breakpoints are specified on a line basis. When a process executing code in an interpreted module reaches a breakpoint, it stops. This means that a breakpoint must be set at an executable line, that is, a code line containing an executable expression.

A breakpoint has the following:

* A status, which is *active* or *inactive*. An inactive breakpoint is ignored.

* A trigger action. When a breakpoint is reached, the trigger action specifies if the breakpoint is to continue as active (*enable*), or to become inactive (*disable*), or to be removed (*delete*).

* Optionally an associated condition. A condition is a tuple *{Module,Name}*. When the breakpoint is reached, *Module:Name(Bindings)* is called. If it evaluates to *true*, execution stops. If it evaluates to *false*, the breakpoint is ignored. *Bindings* contains the current variable bindings. To

retrieve the value for a specified variable, use *get_binding*.

By default, a breakpoint is active, has trigger action *enable*, and has no associated condition. For details about breakpoints, see the User's Guide.

EXPORTS

i(AbsModule) -> {module,Module} | error
i(AbsModules) -> ok
ni(AbsModule) -> {module,Module} | error
ni(AbsModules) -> ok

Types:

```
AbsModules = [AbsModule]
AbsModule = Module | File | [Module | File]
 Module = atom()
 File = string()
```

Interprets the specified module(s). *i/1* interprets the module only at the current node. *ni/1* interprets the module at all known nodes.

A module can be specified by its module name (atom) or filename.

If specified by its module name, the object code *Module.beam* is searched for in the current path. The source code *Module.erl* is searched for first in the same directory as the object code, then in an *src* directory next to it.

If specified by its filename, the filename can include a path and the *.erl* extension can be omitted. The object code *Module.beam* is searched for first in the same directory as the source code, then in an *ebin* directory next to it, and then in the current path.

Note:

The interpreter requires both the source code and the object code. The object code *must* include debug information, that is, only modules compiled with option *debug_info* set can be interpreted.

The functions returns *{module,Module}* if the module was interpreted, otherwise *error* is returned.

The argument can also be a list of modules or filenames, in which case the function tries to interpret each module as specified earlier. The function then always returns *ok*, but prints some information to *stdout* if a module cannot be interpreted.

n(AbsModule) -> ok
nn(AbsModule) -> ok

Types:

AbsModule = Module | File | [Module | File]
Module = atom()
File = string()

Stops interpreting the specified module. *n/1* stops interpreting the module only at the current node. *nn/1* stops interpreting the module at all known nodes.

As for *i/1* and *ni/1*, a module can be specified by its module name or filename.

interpreted() -> [Module]

Types:

Module = atom()

Returns a list with all interpreted modules.

file(Module) -> File | {error,not_loaded}

Types:

Module = atom()
File = string()

Returns the source code filename *File* for an interpreted module *Module*.

interpretable(AbsModule) -> true | {error,Reason}

Types:

AbsModule = Module | File
Module = atom()
File = string()
Reason = no_src | no_beam | no_debug_info | badarg | {app,App}
App = atom()

Checks if a module can be interpreted. The module can be specified by its module name *Module* or its source filename *File*. If specified by a module name, the module is searched for in the code path.

The function returns *true* if all of the following apply:

* Both source code and object code for the module is found.
* The module has been compiled with option *debug_info* set.
* The module does not belong to any of the applications Kernel, STDLIB, GS, or Debugger.

The function returns *{error,Reason}* if the module cannot be interpreted. *Reason* can have the following values:

no_src:
No source code is found. It is assumed that the source code and object code are located either in the same directory, or in *src* and *ebin* directories next to each other.

no_beam:
No object code is found. It is assumed that the source code and object code are located either in the same directory, or in *src* and *ebin* directories next to each other.

no_debug_info:
The module has not been compiled with option *debug_info* set.

badarg:
AbsModule is not found. This could be because the specified file does not exist, or because *code:which/1* does not return a BEAM filename, which is the case not only for non-existing modules but also for modules that are preloaded or cover-compiled.

{app,App}:
App is *kernel*, *stdlib*, *gs*, or *debugger* if *AbsModule* belongs to one of these applications.

Notice that the function can return *true* for a module that in fact is not interpretable in the case where the module is marked as sticky or resides in a directory marked as sticky. The reason is that this is not discovered until the interpreter tries to load the module.

auto_attach() -> false | {Flags,Function}
auto_attach(false)

auto_attach(Flags, Function)

Types:

Flags = [init | break | exit]
Function = {Module,Name,Args}
Module = Name = atom()
Args = [term()]

Gets and sets when and how to attach automatically to a process executing code in interpreted modules. *false* means never attach automatically, this is the default. Otherwise automatic attach is defined by a list of flags and a function. The following flags can be specified:

* *init* - Attach when a process for the first time calls an interpreted function.

* *break* - Attach whenever a process reaches a breakpoint.

* *exit* - Attach when a process terminates.

When the specified event occurs, the function *Function* is called as:

spawn(Module, Name, [Pid | Args])

Pid is the pid of the process executing interpreted code.

stack_trace() -> Flag
stack_trace(Flag)

Types:

Flag = all | no_tail | false

Gets and sets how to save call frames in the stack. Saving call frames makes it possible to inspect the call chain of a process, and is also used to emulate the stack trace if an error (an exception of class error) occurs. The following flags can be specified:

all:
Save information about all current calls, that is, function calls that have not yet returned a value.

no_tail:
Save information about current calls, but discard previous information when a tail recursive call is made. This option consumes less memory and can be necessary to use for processes with long lifetimes and many tail recursive calls. This is the default.

false:
Save no information about current calls.

break(Module, Line) -> ok | {error,break_exists}

Types:

Module = atom()
Line = int()

Creates a breakpoint at *Line* in *Module*.

delete_break(Module, Line) -> ok

Types:

Module = atom()
Line = int()

Deletes the breakpoint at *Line* in *Module*.

break_in(Module, Name, Arity) -> ok | {error,function_not_found}

Types:

Module = Name = atom()
Arity = int()

Creates a breakpoint at the first line of every clause of function *Module:Name/Arity*.

del_break_in(Module, Name, Arity) -> ok | {error,function_not_found}

Types:

Module = Name = atom()

Arity = int()

Deletes the breakpoints at the first line of every clause of function *Module:Name/Arity*.

no_break() -> ok
no_break(Module) -> ok

Deletes all breakpoints, or all breakpoints in *Module*.

disable_break(Module, Line) -> ok

Types:

Module = atom()
Line = int()

Makes the breakpoint at *Line* in *Module* inactive.

enable_break(Module, Line) -> ok

Types:

Module = atom()
Line = int()

Makes the breakpoint at *Line* in *Module* active.

action_at_break(Module, Line, Action) -> ok

Types:

Module = atom()
Line = int()
Action = enable | disable | delete

Sets the trigger action of the breakpoint at *Line* in *Module* to *Action*.

test_at_break(Module, Line, Function) -> ok

Types:

Module = atom()
Line = int()
Function = {Module,Name}
Name = atom()

Sets the conditional test of the breakpoint at *Line* in *Module* to *Function*. The function must fulfill the requirements specified in section Breakpoints.

get_binding(Var, Bindings) -> {value,Value} | unbound

Types:

Var = atom()
Bindings = term()
Value = term()

Retrieves the binding of *Var*. This function is intended to be used by the conditional function of a breakpoint.

all_breaks() -> [Break]
all_breaks(Module) -> [Break]

Types:

Break = {Point,Options}
Point = {Module,Line}
Module = atom()
Line = int()
Options = [Status,Trigger,null,Cond|]
Status = active | inactive
Trigger = enable | disable | delete
Cond = null | Function
Function = {Module,Name}
Name = atom()

Gets all breakpoints, or all breakpoints in *Module*.

snapshot() -> [Snapshot]

Types:

```
Snapshot = {Pid, Function, Status, Info}
 Pid = pid()
 Function = {Module,Name,Args}
 Module = Name = atom()
 Args = [term()]
 Status = idle | running | waiting | break | exit | no_conn
 Info = {} | {Module,Line} | ExitReason
 Line = int()
 ExitReason = term()
```

Gets information about all processes executing interpreted code.

* *Pid* - Process identifier.

* *Function* - First interpreted function called by the process.

* *Status* - Current status of the process.

* *Info* - More information.

Status is one of the following:

* *idle* - The process is no longer executing interpreted code. *Info={}*.

* *running* - The process is running. *Info={}*.

* *waiting* - The process is waiting at a *receive*. *Info={}*.

* *break* - Process execution is stopped, normally at a breakpoint. *Info={Module,Line}*.

* *exit* - The process is terminated. *Info=ExitReason*.

* *no_conn* - The connection is down to the node where the process is running. *Info={}*.

clear() -> ok

Clears information about processes executing interpreted code by removing all information about terminated processes.

continue(Pid) -> ok | {error,not_interpreted}

continue(X,Y,Z) -> ok | {error,not_interpreted}

Types:

Pid = pid()
X = Y = Z = int()

Resumes process execution for *Pid* or *c:pid(X,Y,Z)*.