

Debugger

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Debugger 4.2.4
August 17, 2018

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

August 17, 2018

1 Debugger User's Guide

1.1 Introduction

1.1.1 Scope

Debugger is a graphical user interface for the Erlang interpreter, which can be used for debugging and testing of Erlang programs. For example, breakpoints can be set, code can be single-stepped and variable values can be displayed and changed.

The Erlang interpreter can also be accessed through the interface module *`int(3)`*.

Warning:

Debugger might at some point start tracing on the processes that execute the interpreted code. This means that a conflict occurs if tracing by other means is started on any of these processes.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

Modules to be debugged must include debug information, for example, `erlc +debug_info MODULE.erl`.

1.2 Debugger

1.2.1 Getting Started

To use Debugger, the basic steps are as follows:

Step 1. Start Debugger by calling `debugger:start()`.

The *Monitor window* is displayed with information about all debugged processes, interpreted modules, and selected options. Initially there are normally no debugged processes. First, it must be specified which modules that are to be **debugged** (also called **interpreted**). Proceed as follows:

Step 2. Select **Module** > **Interpret...** in the Monitor window.

The *Interpret Modules window* is displayed.

Step 3. Select the appropriate modules from the Interpret Dialog window.

Note:

Only modules compiled with option `debug_info` set can be interpreted. Non-interpretable modules are displayed within parenthesis in the Interpret Modules window.

Step 4. In the Monitor window, select **Module** > **the module to be interpreted** > **View**.

The contents of the source file is displayed in the *View Module window*.

Step 5. Set the *breakpoints*, if any.

Step 6. Start the program to be debugged. This is done the normal way from the Erlang shell.

All processes executing code in interpreted modules are displayed in the Monitor window.

Step 7. To **attach** to one of these processes, double-click it, or select the process and then choose **Process** > **Attach**. Attaching to a process opens an *Attach Process window* for this process.

Step 8. From the Attach Process window, you can control the process execution, inspect variable values, set breakpoints, and so on.

1.2.2 Breakpoints and Break Dialog Windows

Once the appropriate modules are interpreted, breakpoints can be set at relevant locations in the source code. Breakpoints are specified on a line basis. When a process reaches a breakpoint, it stops and waits for commands (**Step**, **Skip**, **Continue** ...) from the user.

Note:

When a process reaches a breakpoint, only that process is stopped. Other processes are not affected.

Breakpoints are created and deleted using the **Break** menu of either the Monitor window, View Module window, or Attach Process window.

Executable Lines

To have an effect, a breakpoint must be set at an **executable line**, which is a line of code containing an executable expression such as a matching or a function call. A blank line or a line containing a comment, function head, or pattern in a `case` statement or `receive` statement is not executable.

In the following example, lines 2, 4, 6, 8, and 11 are executable lines:

```
1: is_loaded(Module,Compiled) ->
2:   case get_file(Module,Compiled) of
3:     {ok,File} ->
4:       case code:which(Module) of
5:         ?TAG ->
6:           {loaded,File};
7:         _ ->
8:           unloaded
9:       end;
10:    false ->
11:      false
12:  end.
```

Status and Trigger Action

A breakpoint can be either **active** or **inactive**. Inactive breakpoints are ignored.

Each breakpoint has a **trigger action** that specifies what is to happen when a process has reached it (and stopped):

- **Enable** - Breakpoint is to remain active (default).
- **Disable** - Breakpoint is to be made inactive.
- **Delete** - Breakpoint is to be deleted.

Line Breakpoints

A line breakpoint is created at a certain line in a module.

Figure 2.1: Line Break Dialog Window

Right-click the **Module** entry to open a popup menu from which the appropriate module can be selected.

A line breakpoint can also be created (and deleted) by double-clicking the line when the module is displayed in the View Module window or Attach Process window.

Conditional Breakpoints

A conditional breakpoint is created at a certain line in the module, but a process reaching the breakpoint stops only if a specified condition is true.

The condition is specified by the user as a module name `CModule` and a function name `CFunction`. When a process reaches the breakpoint, `CModule:CFunction(Bindings)` is evaluated. If and only if this function call returns `true`, the process stops. If the function call returns `false`, the breakpoint is silently ignored.

`Bindings` is a list of variable bindings. To retrieve the value of `Variable` (given as an atom), use function *`int:get_binding(Variable,Bindings)`*. The function returns `unbound` or `{value,Value}`.

Figure 2.2: Conditional Break Dialog Window

Right-click the **Module** entry to open a popup menu from which the appropriate module can be selected.

Example:

A conditional breakpoint calling `c_test:c_break/1` is added at line 6 in module `fact`. Each time the breakpoint is reached, the function is called. When `N` is equal to 3, the function returns `true` and the process stops.

Extract from `fact.erl`:

```
5. fac(0) -> 1;
6. fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
```

Definition of `c_test:c_break/1`:

```
-module(c_test).
-export([c_break/1]).

c_break(Bindings) ->
    case int:get_binding('N', Bindings) of
        {value, 3} ->
            true;
        _ ->
            false
    end.
```

Function Breakpoints

A function breakpoint is a set of line breakpoints, one at the first line of each clause in the specified function.

Figure 2.3: Function Break Dialog Window

To open a popup menu from which the appropriate module can be selected, right-click the **Module** entry.

To bring up all functions of the module in the listbox, click the **OK** button (or press the **Return** or **Tab** key) when a module name has been specified,.

1.2.3 Stack Trace

The Erlang emulator keeps track of a **stack trace**, information about recent function calls. This information is used if an error occurs, for example:

```
1> catch a+1.
{'EXIT',{badarith,[{erlang,'+',[a,1],[]},
                   {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,573}]},
                   {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,357}]},
                   {shell,exprs,7,[{file,"shell.erl"},{line,674}]},
                   {shell,eval_exprs,7,[{file,"shell.erl"},{line,629}]},
                   {shell,eval_loop,3,[{file,"shell.erl"},{line,614}]}]}}
```

For details about the stack trace, see section *Errors and Error Handling* in the Erlang Reference Manual.

Debugger emulates the stack trace by keeping track of recently called interpreted functions. (The real stack trace cannot be used, as it shows which functions of Debugger have been called, rather than which interpreted functions.)

This information can be used to traverse the chain of function calls, using the **Up** and **Down** buttons in the *Attach Process window*.

By default, Debugger only saves information about recursive function calls, that is, function calls that have not yet returned a value (option **Stack On, No Tail**).

Sometimes, however, it can be useful to save all calls, even tail-recursive calls. This is done with option **Stack On, Tail**. Notice that this option consumes more memory and slows down execution of interpreted functions when there are many tail-recursive calls.

To turn off the Debugger stack trace facility, select option **Stack Off**.

Note:

If an error occurs, the stack trace becomes empty in this case.

For information about how to change the stack trace option, see section *Monitor Window*.

1.2.4 Monitor Window

The Monitor window is the main window of Debugger and displays the following:

- A listbox containing the names of all interpreted modules

  Double-clicking a module brings up the View Module window.
- Which options are selected
- Information about all debugged processes, that is, all processes that have been or are executing code in interpreted modules

Figure 2.4: Monitor Window

The **Auto Attach** boxes, **Stack Trace** label, **Back Trace Size** label, and **Strings** box display some options set. For details about these options, see section *Options Menu*.

Process Grid

Pid

The process identifier.

Initial Call

The first call to an interpreted function by this process. (`Module:Function/Arity`)

Name

The registered name, if any. If a registered name is not displayed, it can be that Debugger received information about the process before the name was registered. Try selecting **Edit** > **Refresh**.

Status

The current status, one of the following:

idle

The interpreted function call has returned a value, and the process is no longer executing interpreted code.

running

The process is running.

waiting

The process is waiting in a `receive` statement.

break

The process is stopped at a breakpoint.

exit

The process has terminated.

no_conn

There is no connection to the node where the process is located.

Information

More information, if any. If the process is stopped at a breakpoint, the field contains information about the location `{Module,Line}`. If the process has terminated, the field contains the exit reason.

File Menu

Load Settings...

Tries to load and restore Debugger settings from a file previously saved using **Save Settings...** (see below). Any errors are silently ignored.

Notice that settings saved by Erlang/OTP R16B01 or later cannot be read by Erlang/OTP R16B or earlier.

Save Settings...

Saves Debugger settings to a file. The settings include the set of interpreted files, breakpoints, and the selected options. The settings can be restored in a later Debugger session using **Load Settings...** (see above). Any errors are silently ignored.

Exit

Stops Debugger.

Edit Menu

Refresh

Updates information about debugged processes. Information about all terminated processes are removed from the window. All Attach Process windows for terminated processes are closed.

Kill All

Terminates all processes listed in the window using `exit(Pid,kill)`.

Module Menu

Interpret...

Opens the *Interpret Modules window*, where new modules to be interpreted can be specified.

Delete All

Stops interpreting all modules. Processes executing in interpreted modules terminate.

For each interpreted module, a corresponding entry is added to the **Module** menu, with the following submenu:

Delete

Stops interpreting the selected module. Processes executing in this module terminate.

View

Opens a *View Module window*, displaying the contents of the selected module.

Process Menu

The following menu items apply to the currently selected process, provided it is stopped at a breakpoint (for details, see section *Attach Process window*):

Step
Next
Continue
Finish

The following menu items apply to the currently selected process:

Attach

Attaches to the process and open an *Attach Process window*.

Kill

Terminates the process using `exit(Pid,kill)`.

Break Menu

The items in this menu are used to create and delete breakpoints. For details, see section *Breakpoints*.

Line Break...

Sets a line breakpoint.

Conditional Break...

Sets a conditional breakpoint.

Function Break...

Sets a function breakpoint.

Enable All

Enables all breakpoints.

Disable All

Disables all breakpoints.

Delete All

Removes all breakpoints.

For each breakpoint, a corresponding entry is added to the **Break** menu, from which it is possible to disable, enable, or delete the breakpoint, and to change its trigger action.

Options Menu

Trace Window

Sets the areas to be visible in an *Attach Process window*. Does not affect existing Attach Process windows.

Auto Attach

Sets the events a debugged process is to be attached to automatically. Affects existing debugged processes.

- **First Call** - The first time a process calls a function in an interpreted module.
- **On Exit** - At process termination.
- **On Break** - When a process reaches a breakpoint.

Stack Trace

Sets the stack trace option, see section *Stack Trace*. Does not affect existing debugged processes.

- **Stack On, Tail** - Saves information about all current calls.
- **Stack On, No Tail** - Saves information about current calls, discarding previous information when a tail recursive call is made.
- **Stack Off** - Does not save any information about current calls.

Strings

Sets the integer lists to be printed as strings. Does not affect existing debugged processes.

- **Use range of +pc flag** - Uses the printable character range set by the `erl(1)` flag `+pc`.

Back Trace Size...

Sets how many call frames to be fetched when inspecting the call stack from the Attach Process window. Does not affect existing Attach Process windows.

Windows Menu

Contains a menu item for each open Debugger window. Selecting one of the items raises the corresponding window.

Help Menu

Help

Shows the Debugger documentation. This function requires a web browser.

1.2.5 Interpret Modules Window

The Interpret Modules window is used for selecting which modules to interpret. Initially, the window displays the modules (`erl` files) and subdirectories of the current working directory.

Interpretable modules are modules for which a `.beam` file, compiled with option `debug_info` set, is located in the same directory as the source code, or in an `ebin` directory next to it.

Modules for which these requirements are not fulfilled are not interpretable and are therefore displayed within parentheses.

Option `debug_info` causes **debug information** or **abstract code** to be added to the `.beam` file. This increases the file size and makes it possible to reconstruct the source code. It is therefore recommended not to include debug information in code aimed for target systems.

An example of how to compile code with debug information using `erlc`:

```
% erlc +debug_info module.erl
```

An example of how to compile code with debug information from the Erlang shell:

```
4> c(module, debug_info).
```

Figure 2.5: Interpret Modules Window

To browse the file hierarchy and interpret the appropriate modules, either select a module name and click **Choose** (or press carriage return), or double-click the module name. Interpreted modules have the type `erl src`.

To interpret all displayed modules in the chosen directory, click **All**.

To close the window, click **Done**.

> **Note:**
>
> When Debugger is started in global mode (which is the default, see *debugger:start/0*), modules added (or deleted) for interpretation are added (or deleted) on all known Erlang nodes.

1.2.6 Attach Process Window

From an Attach Process window, you can interact with a debugged process. One window is opened for each process that has been attached to. Notice that when attaching to a process, its execution is automatically stopped.

Figure 2.6: Attach Process Window

The window is divided into the following five parts:

- The Code area, displaying the code being executed. The code is indented and each line is prefixed with its line number. If the process execution is stopped, the current line is marked with `-->`. An existing break point at a line is marked with a stop symbol. In the example shown in the illustration, the execution stopped at line 6, before the execution of `fac/1`.

  Active breakpoints are displayed in red and inactive breakpoints in blue.
- The Button area, with buttons for quick access to frequently used functions in the **Process** menu.
- The Evaluator area, where you can evaluate functions within the context of the debugged process, if that process execution is stopped.
- The Bindings area, displaying all variables bindings. If you click a variable name, the value is displayed in the Evaluator area. Double-click a variable name to open a window where the variable value can be edited. Notice however that pid, port, reference, or fun values cannot be edited unless they can be represented in the running system.
- The Trace area, which displays a trace output for the process.

  `++ (N) <L>`

  Function call, where `N` is the call level and `L` the line number.

  `-- (N)`

  Function return value

.

`==> Pid : Msg`

The message `Msg` is sent to process `Pid`.

`<== Msg`

The message `Msg` is received.

`++ (N) receive`

Waiting in a `receive`.

`++ (N) receive with timeout`

Waiting in a `receive...after`.

The Trace area also displays Back Trace, a summary of the current function calls on the stack.

Using the **Options** menu, you can set which areas to be displayed. By default, all areas except the Trace area are displayed.

File Menu

Close

Closes this window and detach from the process.

Edit Menu

Go to line...

Goes to a specified line number.

Search...

Searches for a specified string.

Process Menu

Step

Executes the current code line, stepping into any (interpreted) function calls.

Next

Executes the current code line and stop at the next line.

Continue

Continues the execution.

Finish

Continues the execution until the current function returns.

Skip

Skips the current code line and stop at the next line. If used on the last line in a function body, the function returns `skipped`.

Time Out

Simulates a time-out when executing a `receive...after` statement.

Stop

Stops the execution of a running process, that is, make the process stop at a breakpoint. The command takes effect (visibly) the next time the process receives a message.

Where

Verifies that the current location of the execution is visible in the code area.

Kill

Terminates the process using `exit(Pid,kill)`.

Messages

Inspects the message queue of the process. The queue is displayed in the Evaluator area.

Back Trace

Displays the back trace of the process, a summary of the current function calls on the stack, in the Trace area. Requires that the Trace area is visible and that the Stack Trace option is **Stack On, Tail** or **Stack On, No Tail**.

Up

Inspects the previous function call on the stack, showing the location and variable bindings.

Down

Inspects the next function call on the stack, showing the location and variable bindings.

Options Menu

Trace Window

Sets which areas are to be visible. Does not affect other Attach Process windows.

Stack Trace

Same as in the *Monitor window*, but only affects the debugged process the window is attached to.

Strings

Same as in the *Monitor window*, but only affects the debugged process the window is attached to.

Back Trace Size...

Sets how many call frames are to be fetched when inspecting the call stack. Does not affect other Attach Process windows.

Break, Windows, and Help Menus

The **Break**, **Windows**, and **Help** menus are the same as in the *Monitor Window*, except that the **Breaks** menu applies only to local breakpoints.

1.2.7 View Module Window

The View Module window displays the contents of an interpreted module and makes it possible to set breakpoints.

Figure 2.7: View Module Window

The source code is indented and each line is prefixed with its line number.

Clicking a line highlights it and selects it to be the target of the breakpoint functions available from the **Break** menu. To set a line breakpoint on a line, double-click it. To remove the breakpoint, double-click the line with an existing breakpoint.

Breakpoints are marked with a stop symbol.

File and Edit Menus

The **File** and **Edit** menus are the same as in the *Attach Process Window*.

Break, Windows, and Help Menus

The **Break**, **Windows**, and **Help** menus are the same as in the *Monitor Window*, except that the **Break** menu applies only to local breakpoints.

1.2.8 Performance

Execution of interpreted code is naturally slower than for regularly compiled modules. Using Debugger also increases the number of processes in the system, as for each debugged process another process (the meta process) is created.

It is also worth to keep in mind that programs with timers can behave differently when debugged. This is especially true when stopping the execution of a process (for example, at a breakpoint). Time-outs can then occur in other processes that continue execution as normal.

1.2.9 Code Loading Mechanism

Code loading works almost as usual, except that interpreted modules are also stored in a database and debugged processes use only this stored code. Reinterpreting an interpreted module results in the new version being stored as well, but does not affect existing processes executing an older version of the code. This means that the code replacement mechanism of Erlang does not work for debugged processes.

1.2.10 Debugging Remote Nodes

By using *debugger:start/1*, you can specify if Debugger is to be started in local or global mode:

```
debugger:start(local | global)
```

If no argument is provided, Debugger starts in global mode.

In local mode, code is interpreted only at the current node. In global mode, code is interpreted at all known nodes. Processes at other nodes executing interpreted code are automatically displayed in the Monitor window and can be attached to like any other debugged process.

It is possible, but definitely not recommended, to start Debugger in global mode on more than one node in a network, as the nodes interfere with each other, leading to inconsistent behavior.

2 Reference Manual

debugger

Erlang module

Erlang Debugger for debugging and testing of Erlang programs.

Exports

start()
start(File)
start(Mode)
start(Mode, File)

Types:

Mode = local | global
File = string()

Starts Debugger.

If a filename is specified as argument, Debugger tries to load its settings from this file. For details about settings, see the User's Guide.

If `local` is specified as argument, Debugger interprets code only at the current node. If `global` is specified as argument, Debugger interprets code at all known nodes, this is the default.

quick(Module, Name, Args)

Types:

Module = Name = atom()
Args = [term()]

Debugs a single process. The module `Module` is interpreted and `apply(Module,Name,Args)` is called. This opens an Attach Process window. For details, see the User's Guide.

i

Erlang module

The `i` module provides short forms for some of the functions used by the graphical Debugger and some of the functions in module *`int`*, the Erlang interpreter.

This module also provides facilities for displaying status information about interpreted processes and break points.

It is possible to attach to interpreted processes by giving the corresponding process identity only. By default, an attachment window is displayed. Processes at other Erlang nodes can be attached manually or automatically.

By preference, these functions can be included in module *`shell_default`*. By default, they are included in that module.

Exports

`im() -> pid()`

Starts a new graphical monitor. This is the Monitor window, the main window of Debugger. All the Debugger and interpreter functionality is accessed from the Monitor window. This window displays the status of all processes that have been or are executing interpreted modules.

```
ii(AbsModules) -> ok
ii(AbsModule) -> {module, Module} | error
ini(AbsModules) -> ok
ini(AbsModule) -> {module, Module} | error
```

Types:

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

Interprets the specified module(s). `ii/1` interprets the module(s) only at the current node, see *int:i/1*. `ini/1` interprets the module(s) at all known nodes, see *int:ni/1*.

```
iq(AbsModule) -> ok
inq(AbsModule) -> ok
```

Types:

```
AbsModule = Module | File
 Module = atom()
 File = string()
```

Stops interpreting the specified module. `iq/1` stops interpreting the module only at the current node. `inq/1` stops interpreting the module at all known nodes.

`il() -> ok`

Makes a printout of all interpreted modules. Modules are printed together with the full path name of the corresponding source code file.

ip() -> ok

Prints the current status of all interpreted processes.

ic() -> ok

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

iaa(Flags) -> true
iaa(Flags, Function) -> true

Types:

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

Sets when and how to attach to a debugged process automatically, see *int:auto_attach/2*. `Function` defaults to the standard function used by Debugger.

ist(Flag) -> true

Types:

```
Flag = all | no_tail | false
```

Sets how to save call frames in the stack, see *int:stack_trace/1*.

ia(Pid) -> ok | no_proc

Types:

```
Pid = pid()
```

Attaches to the debugged process `Pid`. An Attach Process window is opened for the process.

ia(X,Y,Z) -> ok | no_proc

Types:

```
X = Y = Z = int()
```

Same as `ia(Pid)`, where `Pid` is the result of calling the shell function `pid(X,Y,Z)`.

ia(Pid, Function) -> ok | no_proc

Types:

```
Pid = pid()
Function = {Module,Name}
 Module = Name = atom()
```

Attaches to the debugged process `Pid`. The interpreter calls `spawn(Module, Name, [Pid])` (and ignores the result).

ia(X,Y,Z, Function) -> ok | no_proc

Types:

```
X = Y = Z = int()
Function = {Module,Name}
```

Module = Name = atom()

Same as `ia(Pid, Function)`, where `Pid` is the result of calling the shell function `pid(X,Y,Z)`. An attached process is expected to call the unofficial function `int:attached(Pid)` and to be able to handle messages from the interpreter. For an example, see `dbg_wx_trace.erl`.

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

Types:

Module = atom()
Line = int()

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

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

Types:

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

Creates breakpoints at the first line of every clause of function `Module:Name/Arity`.

ir() -> ok

Deletes all breakpoints.

ir(Module) -> ok

Types:

Module = atom()

Deletes all breakpoints in `Module`.

ir(Module, Line) -> ok

Types:

Module = atom()
Line = int()

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

ir(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`.

ibd(Module, Line) -> ok

Types:

Module = atom()
Line = int()

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

ibe(Module, Line) -> ok

Types:

Module = atom()
Line = int()

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

iba(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`.

ibc(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 conditional test is performed by calling `Module:Name(Bindings)`, where `Bindings` is the current variable bindings. The function must return `true` (break) or `false` (do not break). To retrieve the value of a variable `Var`, use *int:get_binding(Var, Bindings)*.

ipb() -> ok

Prints all existing breakpoints.

ipb(Module) -> ok

Types:

Module = atom()

Prints all existing breakpoints in `Module`.

iv() -> atom()

Returns the current version number of the interpreter. Same as the version number of the Debugger application.

help() -> ok

Prints help text.

See Also

int(3)

int

Erlang module

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 currentcalls.

`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)`.