Debugger Application (DEBUGGER)

version 2.3

Contents

Debugger Application (DEBUGGER)

Chapter 1

Debugger User's Guide

Debugger is a graphical tool 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.

1.1 Debugger

1.1.1 Introduction

Debugger is a graphical tool 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.

Warning: Note that the debugger at some point might start tracing on the processes which execute the interpreted code. This means that a conflict will occur if tracing by other means is started on any of these processes.

1.1.2 Getting Started with Debugger

Start Debugger by calling `debugger:start()`. It will start the monitor window [page 5] showing information about all debugged processes.

Initially there are normally no debugged processes. First, it must be specified which modules should be *debugged*, or *interpreted* as it is also called. This is done by chosing *Module->Interpret...* in the monitor window and then selecting the appropriate modules from the interpret dialog window [page 9].

> **Note:**
> Only modules compiled with the option `debug_info` set can be interpreted. Non-interpretable modules are shown within parenthesis in the interpret dialog window.

When a module is interpreted, it can be viewed in a view module window [page 14]. This is done by selecting the module from the *Module->module->View* menu. The contents of the source file is shown and it is possible to set breakpoints [page 2].

Now the program that should be debugged can be started. This is done the normal way from the Erlang shell. All processes executing code in interpreted modules will be displayed in the monitor window. It is possible to *attach* to any of these processes, by selecting the process and then chosing *Process->Attach*.

Attaching to a process will result in a attach process window [page 11] being opened for this process. From the attach process window, it is possible to control the process execution, inspect variable values, set breakpoints etc.

1.1.3 Breakpoints

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 the monitor window, view module window and attach process window.

Executable Lines

To have 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-` or `receive` statement is not executable.

In the example below, lines number 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* which specifies what should happen when a process has reached it (and stopped):

- *enable* Breakpoint should remain active (default).
- *disable* Breakpoint should be made inactive.
- *delete* Breakpoint should be deleted.

Line Breakpoints

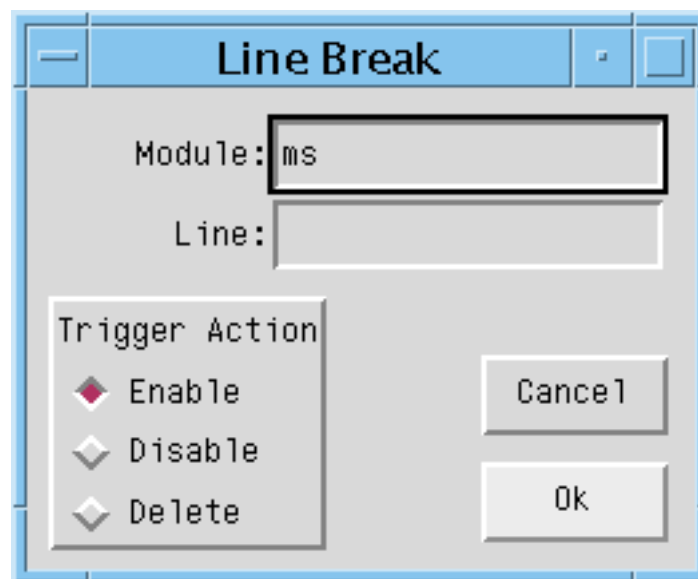A line breakpoint is created at a certain line in a module.

Figure 1.1: The Line Break Dialog Window.

Conditional Breakpoints

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

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

`Bindings` is a list of variable bindings. Use the function `int:get_binding(Variable,Bindings)` to retrieve the value of `Variable` (given as an atom). The function returns `unbound` or `{value,Value}`.
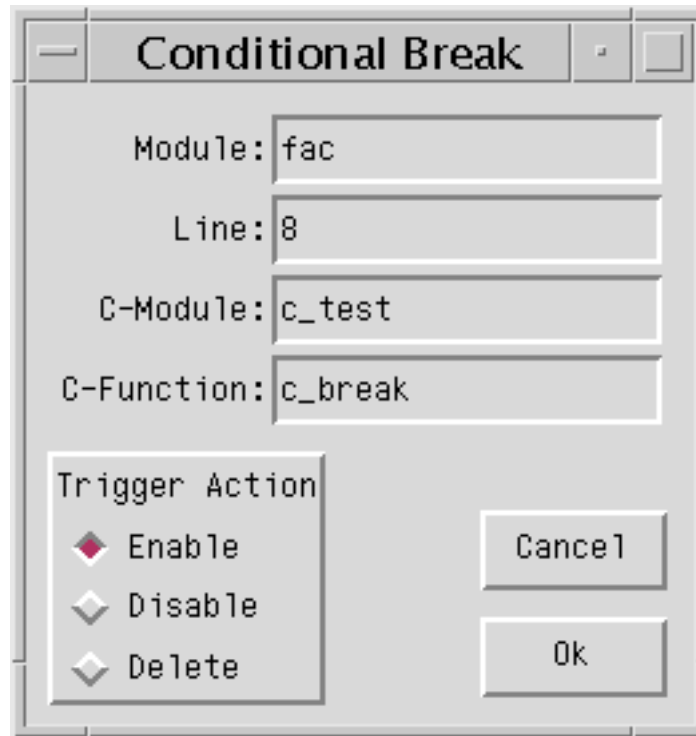
Figure 1.2: The Conditional Break Dialog Window.

Example: A conditional breakpoint calling `c_test:c_break/1` is added at line 8 in the module `fac`. Each time the breakpoint is reached, the function is called, and when `N` is equal to 3 it returns `true`, and the process stops.

Extract from `fac.erl`:

```
4.    fac(0) ->
5.        1;
6.
7.    fac(N) ->
8.        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 given function.
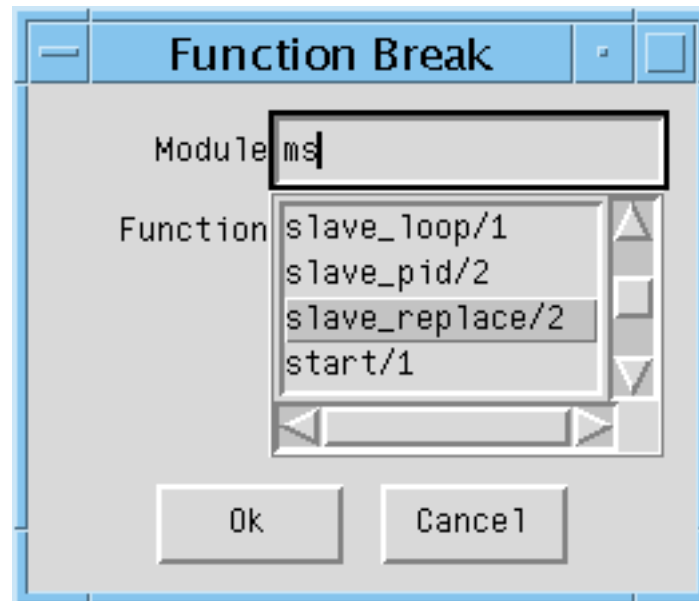

Figure 1.3: The Function Break Dialog Window.

1.1.4   The Monitor Window

The monitor window is the main window of Debugger and shows information about all debugged processes, that is all processes executing code in interpreted modules.
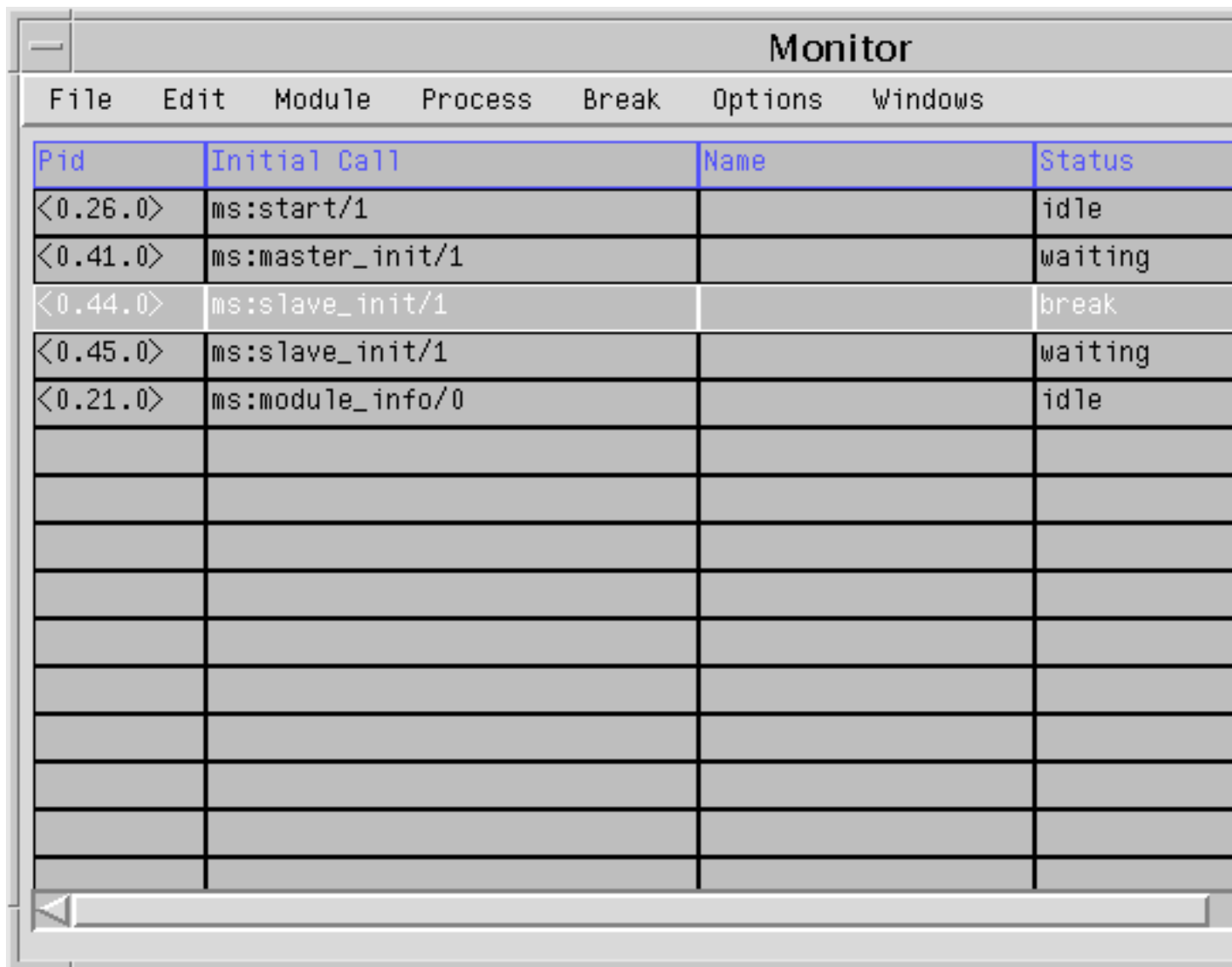
Figure 1.4: The Monitor Window.

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.

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 Additional 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.

The File Menu

Load Settings... Try to load and restore Debugger settings from a file previously saved using *Save Settings...*, see below. Any errors are silently ignored.

Save Settings... Save 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 Stop Debugger.

The Edit Menu

Clear Remove information about all terminated processes from the window.

Kill All Terminate all processes listed in the window using exit(Pid,kill).

The Module Menu

Interpret... Open the interpret dialog window [page 9] where new modules to be interpreted can be specified.

Delete All Stop interpreting all modules. Processes executing in interpreted modules will terminate.

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

Delete Stop interpreting the selected module. Processes executing in this module will terminate.

View Open a view module window [page 14] showing the contents of the selected module.

The Process Menu

The following menu items apply to the currently selected process, provided it is stopped at a breakpoint. See the chapter about the attach process window [page 11] for more information.

Step

Next

Continue

Finish

The following menu items apply to the currently selected process.

Attach Attach to the process and open a attach process window [page 11].

Kill Terminate the process using exit(Pid,kill).

The Break Menu

The items in this menu are used to create and delete breakpoints. See the Breakpoints [page 2] chapter for more information.

Line Break... Set a line breakpoint.

Conditional Break... Set a conditional breakpoint.

Function Break... Set a function breakpoint.

Delete All Remove 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.

The Options Menu

Trace Window Set which areas should be visible in the attach process window [page 11]. Does not affect already existing windows.

Auto Attach Set at which events a debugged process should be automatically attached to. Also affects existing 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 **_Stack On, Tail_** Save call frames in the stack during evaluation.

   Stack On, No Tail Save call frames during evaluation, except for tail recursive calls.

     This option consumes less memory than the previous option and may be necessary to use for processes with long lifetimes and many tail recursive calls.

   Stack Off Do not save call frames.

   Does not affect already existing processes.

Back Trace Size... Set how many call frames should be fetched when inspecting the back trace (from the attach process window). Does not affect already existing windows.

The Windows Menu

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

The Help Menu

Help View the Debugger documentation. Currently this function requires Netscape to be up and running.

1.1.5 The Interpret Dialog Window

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

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

Modules, for which the above requirements are not fulfilled, are not interpretable and are therefore displayed within parentheses.

The `debug_info` option causes *debug information* or *abstract code* to be added to the BEAM file. This will increase the size of the file, and also 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).
```
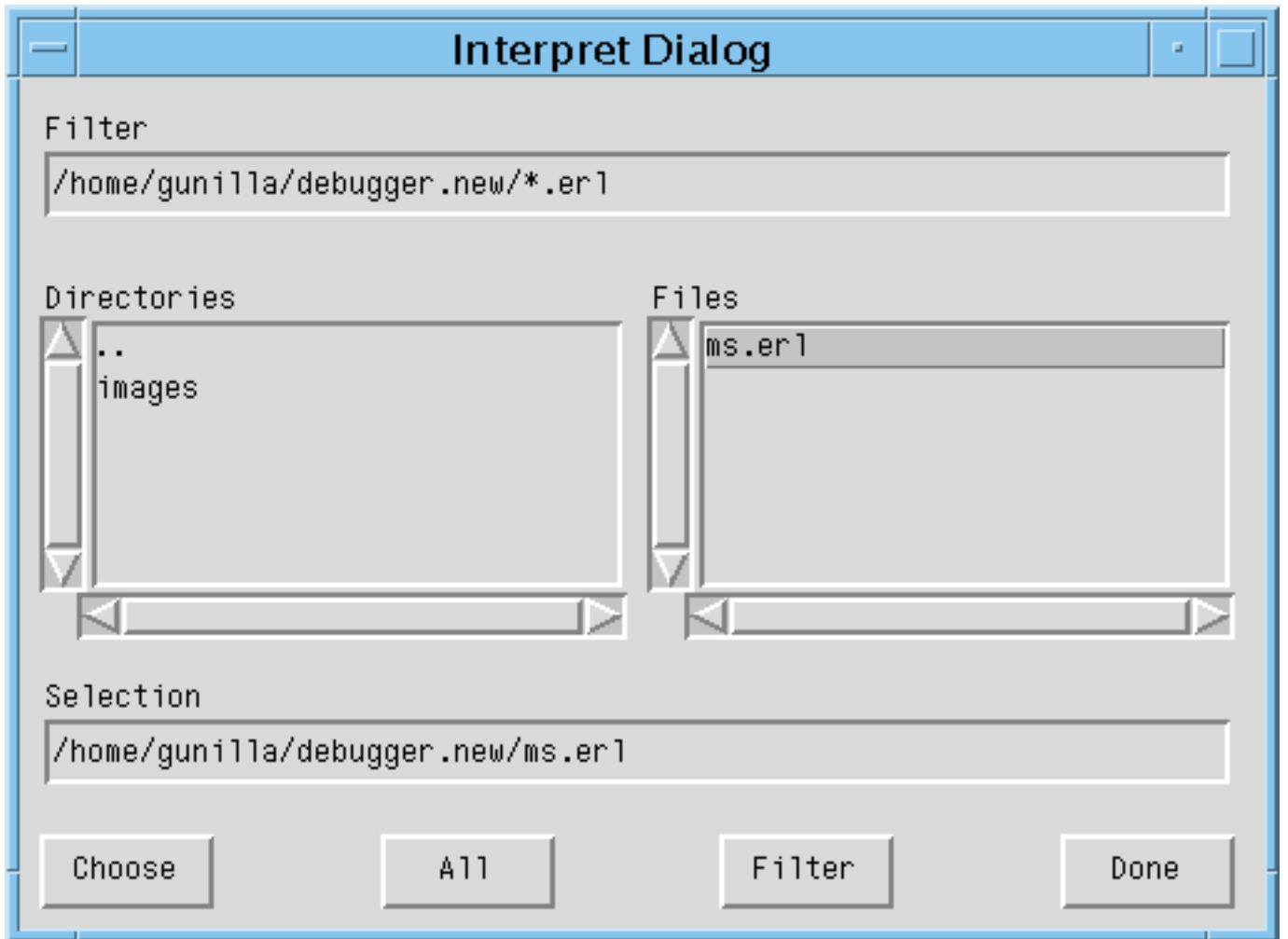
Figure 1.5: The Interpret Dialog Window.

Browse the file hierarchy and interpret the appropriate modules by selecting a module name and pressing *Choose* (or carriage return), or by double clicking the module name. Interpreted modules are displayed with a * in front of the name.

Pressing *All* will interpret all displayed modules in the chosen directory.

Pressing *Done* will close the window.

> **Note:**
> In a distributed environment modules added (or deleted) for interpretation will be added (or deleted) on all known Erlang nodes.

1.1.6   The Attach Process Window

From an attach process window the user can interact with a debugged process. One window is opened for each process that has been attached to. Note that when attaching to a process, its execution is automatically stopped.

```
Attach Process <0.44.0>

File    Edit    Process    Break    Options    Windows                Help

State: break [ms.erl/56]

47:
48:      slave_init(N) ->
49:          slave_loop(N).
50:
51:      slave_loop(N) ->
52:          receive
53:              die ->
54:                  ok;
55:              Msg ->
56:    -->              io:format("Slave ~p got message ~p~n", [N, Msg
57:                  slave_loop(N)
58:          end.
59:
60:      slave_pid(N, [#slave{pid=Pid, number=N}|_Rest]) ->
61:          Pid;
62:      slave_pid(N, [_Slave|Rest]) ->
63:          slave_pid(N, Rest).
64:
65:      slave_replace(Pid, [#slave{pid=Pid, number=N}|Rest]) ->
66:          io:format("master restarting dead slave ~p~n", [N]),
67:          [#slave{pid=spawn_link(ms, slave_init, [N]), number=N}|Rest];
68:      slave_replace(Pid, [Slave | Rest]) ->
69:          [Slave | slave_replace(Pid, Rest)].
```

```
Next    Step    Finish   Continue    Up    Down    Where    Break
```

Evaluator:

Name	Value
Msg	hello
N	2

The window is divided into five parts:

- The code area, showing 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 -@-. In the example above, the execution has been stopped at line 56, before the execution of `io:format/2`.
  Active breakpoints are shown in red, while inactive breakpoints are shown in blue.
- The button area, with buttons for quick access to frequently used functions in the Process menu.
- The evaluator area, where the user can evaluate functions within the context of the debugged process, provided that process execution has been stopped.
- The bindings area, showing all variables bindings. Clicking on a variable name will result in the value being displayed in the evaluator area. Double-clicking on a variable name will open a window where the variable value may be edited. Note however that pid, reference, binary or port values can not be edited.
- The trace area, showing a trace output for the process.

  `++ (N) <L>` Function call, where `N` is the level of recursion and `L` the line number.

  `-- (N)` Function return value. *Note:* Return values of calls with last call optimization are not traced.

  `==> 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`.

  Also the back trace, i.e. a summary of the call frames in the stack, is displayed in the trace area.

It is configurable using the Options menu which areas should be shown or hidden. By default, all areas except the trace area is shown.

The File Menu

Close Close this window and detach from the process.

The Edit Menu

Go to line... Go to a specified line number.

Search... Search for a specified string.

The Process Menu

Step Execute the current line of code, stepping into any function calls.

Next Execute the current line of code and stop at the next line.

Continue Continue the execution.

Finish Continue the execution until the current function returns.

Skip Skip the current line of code and stop at the next line. If used on the last line in a function body, the function will return `skipped`.

Time Out Simulate a timeout when executing a `receive...after` statement.

Stop Stop the execution of a running process.

Where Make sure the current location of the execution is visible in the code area.

Kill Terminate the process using `exit(Pid,kill)`.

Messages Inspect the message queue of the process. The queue is printed in the evaluator area.

Back Trace Display the back trace of the process, i.e. a summary of the call frames on the stack, in the trace area. Requires that the trace area is visible and that the stack trace option is set to 'Stack On, Tail' or 'Stack On, No Tail'.

Up Inspect the previous call frame. The code area will show the location and the bindings area will show the bindings for that call frame.

Down Inspect the next call frame. The code area will show the location and the bindings area will show the bindings for that call frame.

The Options Menu

Trace Window Set which areas should be visible. Does not affect other attach process windows.

Stack Trace Same as in the monitor window [page 5], but does only affect the debugged process the window is attached to.

Back Trace Size... Set how many call frames should be fetched when inspecting the back trace. Does not affect other attach process windows.

Break, Windows and Help Menus

See the chapter The Monitor Window [page 5].

1.1.7   The View Module Window

The view module window shows the contents of an interpreted module and makes it possible to set breakpoints.

```
View Module ms

 File   Edit   Break   Windows


11:
12:       -export([start/1, stop/0, to_slave/2]).
13:       -export([master_init/1, slave_init/1]).
14:
15:       start(N) ->
16:           spawn(ms, master_init, [N]).
17:
18:       stop() ->
19:           master ! stop.
20:
21:       to_slave(Msg, N) ->
22:           master ! {to_slave, self(), Msg, N}.
23:
24:       master_init(N) ->
25:           register(master, self()),
26:           process_flag(trap_exit, true),
27:           master_loop(start_slaves(N, [])).
28:
29:       master_loop(Slaves) ->
30:           receive
31:               stop ->
32:                   exit(stopped);
33:               {to_slave, _Pid, Msg, N} ->
34:                   slave_pid(N, Slaves) ! Msg,
```

Figure 1.7: The View Module Window.

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

Clicking a line will highlight it and select it to be the target of the breakpoint functions available from the Break menu. Doubleclicking a line will set a line breakpoint on that line. Doubleclicking a line with an existing breakpoint will remove the breakpoint.

Breakpoints are marked with -@-.

The Break Menu

The Break menu looks the same as the Break menu in the monitor window, see the chapter The Monitor Window [page 5], except that only breakpoints in the viewed module are shown.

File and Edit Menus

See the chapter The Attach Process Window [page 11].

Windows and Help Menus

See the chapter The Monitor Window [page 5].

1.1.8 Miscellaneous

Performance

Execution of interpreted code is naturally slower than for regularly compiled modules. Using the 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 may behave differently when debugged. This is especially true when stopping the execution of a process, for example at a breakpoint. Timeouts can then occur in other processes that continue execution as normal.

Code loading mechanism

Code loading works almost as usual, except that interpreted modules are also stored in a database and debugged processes uses only this stored code. Re-interpreting an interpreted module will result 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.

1.1.9 Debugging Remote Nodes

By using `debugger:start/1`, it can be specified if Debugger should be started in local or global mode.

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

If no argument is provided, Debugger is started 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 will automatically be shown in the monitor window and can be attached to like any other debugged process.

It is not recommended to use Debugger in global mode on more than one node in a network, as they may interfere with each other leading to inconsistent behaviour.

Debugger Reference Manual

Short Summaries

- Erlang Module **debugger** [page 21] – Erlang Debugger
- Erlang Module **i** [page 22] – Interpreter (debugger) Interface.
- Erlang Module **int** [page 28] – Interpreter Interface.

debugger

The following functions are exported:

- `start()`
  [page 21] Start Debugger.
- `start(File)`
  [page 21] Start Debugger.
- `start(Mode)`
  [page 21] Start Debugger.
- `start(Mode, File)`
  [page 21] Start Debugger.
- `quick(Module, Name, Args)`
  [page 21] Debug a process.

i

The following functions are exported:

- `im() -> pid()`
  [page 22] Start a graphical monitor
- `ii(AbsModule) -> {module, Module} | error`
  [page 22] Interpret a module
- `iq(Module) -> ok`
  [page 23] Do not interpret a module
- `ini(Module) -> {module,Module} | error | ok`
  [page 23] Do not interpret a module on all nodes in the network
- `inq(Module) -> ok`
  [page 23] Do not interpret a module on all nodes in the network
- `il() -> ok`
  [page 23] Make a printout of all interpreted modules

- `ip() -> ok`
  [page 23] Make a printout of the current status of all interpreted processes
- `ic() -> ok`
  [page 23] Delete (clear) information about all terminated processes from the interpreter.
- `iaa(Flag) -> true`
  [page 23] Automatically attaches to a process
- `iaa(Flag,Function) -> true`
  [page 23] Automatically attach to a process using `Function` as the attachment method.
- `ist(Flag) -> true`
  [page 24] Set the usage of call frame inspections
- `ia(Pid) -> ok | no_proc`
  [page 24] Attaches to a process
- `ia(X,Y,Z) -> ok | no_proc`
  [page 24] Attaches to a process
- `ia(Pid,Function) -> ok | no_proc`
  [page 24] Attaches to a process
- `ia(X,Y,Z,Function) -> ok | no_proc`
  [page 24] Attaches to a process
- `ib(Module,Line) -> ok | {error, What}`
  [page 25] Set a break point
- `ib(Module,Function,Arity) -> ok | {error, What}`
  [page 25] Set a break point in a function
- `ir(Module,Line) -> ok | {error, What}`
  [page 25] Delete a break point
- `ir() -> ok`
  [page 25] Delete all existing break points
- `ir(Module) -> ok`
  [page 25] Delete all break points in a module
- `ir(Module,Function,Arity) -> ok | {error, What}`
  [page 25] Deletes a break point in a function
- `ibd(Module,Line) -> ok | {error, What}`
  [page 26] Disable a break point
- `ibe(Module,Line) -> ok | {error, What}`
  [page 26] Enable a break point
- `iba(Module,Line,Action) -> ok | {error, What}`
  [page 26] Set a break point trigger
- `ibc(Module,Line,Function) -> ok | {error, What}`
  [page 26] Set a conditional test at a break point
- `ipb() -> ok`
  [page 26] Make a printout of all existing break points
- `ipb(Module) -> ok`
  [page 27] Make a printout of all break points in module
- `iv() -> atom()`
  [page 27] >Return the current version number of the interpreter
- `help() -> ok`
  [page 27] Print help text

int

The following functions are exported:

- `i(AbsModule) -> {module,Module} | error`
  [page 29] Interpret a module.

- `ni(AbsModule) -> {module,Module} | error`
  [page 29] Interpret a module.

- `n(AbsModule) -> ok`
  [page 29] Stop interpreting a module.

- `nn(AbsModule) -> ok`
  [page 29] Stop interpreting a module.

- `interpreted() -> [Module]`
  [page 29] Get all interpreted modules.

- `file(Module) -> File | {error,not_loaded}`
  [page 29] Get the file name for an interpreted module.

- `interpretable(AbsModule) -> true | {error,Reason}`
  [page 30] Check if a module is possible to interpret.

- `auto_attach() -> false | {Flags,Function}`
  [page 30] Get/set when and how to attach to a process.

- `auto_attach(false)`
  [page 30] Get/set when and how to attach to a process.

- `auto_attach(Flags, Function)`
  [page 30] Get/set when and how to attach to a process.

- `stack_trace() -> Flag`
  [page 30] Get/set if and how to save call frames.

- `stack_trace(Flag)`
  [page 30] Get/set if and how to save call frames.

- `break(Module, Line) -> ok | {error,break_exists}`
  [page 31] Create a breakpoint.

- `delete_break(Module, Line) -> ok`
  [page 31] Delete a breakpoint.

- `break_in(Module, Name, Arity) -> ok | {error,function_not_found}`
  [page 31] Create breakpoints in the specified function.

- `del_break_in(Module, Name, Arity) -> ok | {error,function_not_found}`
  [page 31] Delete breakpoints from the specified function.

- `no_break() -> ok`
  [page 31] Delete all breakpoints.

- `no_break(Module) -> ok`
  [page 31] Delete all breakpoints.

- `disable_break(Module, Line) -> ok`
  [page 31] Make a breakpoint inactive.

- `enable_break(Module, Line) -> ok`
  [page 32] Make a breakpoint active.

- `action_at_break(Module, Line, Action) -> ok`
  [page 32] Set the trigger action of a breakpoint.

- `test_at_break(Module, Line, Function) -> ok`
  [page 32] Set the condition of a breakpoint.

- `get_binding(Var, Bindings) -> {value,Value} | unbound`
  [page 32] Retrieve a variable binding.

- `all_breaks() -> [Break]`
  [page 32] Get all breakpoints.

- `all_breaks(Module) -> [Break]`
  [page 32] Get all breakpoints.

- `snapshot() -> [Snapshot]`
  [page 33] Get information about all processes excuting interpreted code.

- `clear() -> ok`
  [page 33] Clear information about processes executing interpreted code.

- `continue(Pid) -> ok | {error,not_interpreted}`
  [page 33] Resume process execution.

- `continue(X,Y,Z) -> ok | {error,not_interpreted}`
  [page 33] Resume process execution.

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 given a file name as argument, Debugger will try to load its settings from this file. Refer to Debugger User's Guide for more information about settings.
>
> If given `local` as argument, Debugger will interpret code only at the current node. If given `global` as argument, Debugger will interpret code at all known nodes, this is the default.

```
quick(Module, Name, Args)
```

>   Types:
> - Module = Name = atom()
> - Args = [term()]
>
> This function can be used to debug a single process. The module `Module` is interpreted and `apply(Module,Name,Args)` is called. This will open an Attach Process window, refer to Debugger User's Guide for more information.

i

Erlang Module

The module `i` provides short forms for some of the functions in the `int` module.

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

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

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

Exports

`im() -> pid()`

> Starts a new graphical monitor. This is the main window of the interpreter. All of the interpreter functionality is accessed from the monitor window. The monitor window displays the status of all processes that are running interpreted modules.

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

> Types:
> - AbsModule = atom() | string() | [atom() | string()]
> - Module = atom()
>
> Marks `Module` as being interpreted. The `Module` parameter can either be a single module name, or a list of module names. `Module` is compiled into an abstract form which is loaded into the interpreter. The actual paths are searched for the corresponding source file(s) (`Module.erl`). `Module` can be given with an absolute path.
>
> **Note:**
> If `Module` is a list of modules, the result of the last module is returned.
>
> **Note:**
> If an interpreted module is compiled using the c:c function, this module is reloaded into the interpreter.

```
iq(Module) -> ok
```

>   Types:

>   • Module = atom() | string() | [atom() | string()]

>   Does not interpret `Module`. The `Module` parameter can either be a single module name, or a list of module names. `Module` is removed from the set of modules currently being interpreted.

```
ini(Module) -> {module,Module} | error | ok
inq(Module) -> ok
```

>   Behaves as the corresponding `ii/1` and `iq/1` functions described above, but on all nodes in the network. It returns `ok` if we are `alive`, otherwise as above.

```
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
```

>   Makes a printout of the current status of all interpreted processes. Processes on all known nodes are printed.

```
ic() -> ok
```

>   Deletes (clears) information about all terminated processes from the interpreter.

```
iaa(Flag) -> true
```

>   Types:

>   • Flag = FlagItem | [FlagItem]
>   • FlagItem = init | break | exit | false

>   Interpreted processes can be attached automatically, without the need to attach to a process using the monitor window, `i:im()` or `int:m()`. An attachment window - not described here - pops up for the attached process. `Flag` specifies at which point interpreted processes are automatically attached.

>   `Flag` is one of:

>   • `init`. Attach to a process the very first time it calls an interpreted function.
>   • `break`. Attach to a process whenever it reaches a break point.
>   • `exit`. Attach to a process when it terminates.
>   • `false`. Deactivate the automatic attach facility.

>   If several conditions are to be active, a list of flags must be given.

```
iaa(Flag,Function) -> true
```

>   Types:

>   • Flag = FlagItem | [FlagItem]
>   • FlagItem = init | break | exit | false
>   • Function = {Mod,Func}

- Mod = atom()
- Fun = atom()

As above, but instead of using the default attachment window, the specified `Function` is used in order to start the interaction with the attached process. The `Function` parameter must be the tuple {`Mod,Func`}, and this function should implement the corresponding functionality in the same way as the `int show:a start/3,4` functions.

`ist(Flag) -> true`

> Types:

- Flag = all | true | no_tail | false

The interpreter can keep call frames in the stack for future inspections. Typically, you can go up and down in the stack in order to inspect the flow of control when the execution has been stopped - at a break point, when the process has terminated, or in a single step execution.

By default, the whole stack is kept (`Flag` = `all` or `true`). If processes with a very long life time and with a lot of tail recursive calls are interpreted, the `no tail` flag should be used. No tail recursive calls are kept in the stack if this flag is used.

The `false` flag should be used if the interpreter is not to keep call frames.

`ia(Pid) -> ok | no proc`

> Types:

- Pid = pid()

Attaches to the `Pid` process. An attachment window pops up.

`ia(X,Y,Z) -> ok | no proc`

> Types:

- X = Y = Z = int()

Attaches to the process with process identity c:pid(`X,Y,Z`). An attachment window pops up.

`ia(Pid,Function) -> ok | no proc`

> Types:

- Pid = pid()
- Function = {Mod, Fun}
- Mod = atom()
- Fun = atom()

Attaches to the `Pid` process. Use `Function` for the interaction with the attached process, as for the `i:iaa/2` function.

`ia(X,Y,Z,Function) -> ok | no proc`

> Types:

- X = Y = Z = int()
- Function = {Mod, Fun}
- Mod = atom()

- Fun = atom()

Attaches to the process with process identity c:pid(X,Y,Z). Use `Function` for the interaction with the attached process, as for the `i:iaa/2` function.

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

Types:

- Module = atom()
- Line = int()
- What = badarg | break_exists

Creates a new break point at `Line` in `Module`. The execution of an interpreted process will be stopped before the expression at `Line` in `Module` is executed.

ib(Module,Function,Arity) -> ok | {error, What}

Types:

- Module = atom()
- Function = atom()
- Arity = int()
- What = badarg | function_not_found

Creates break points at the first line in every clause of the `Module:Function/Arity` function.

ir(Module,Line) -> ok | {error, What}

Types:

- Module = atom()
- Line = int()
- What = badarg | no_break_exists

Deletes the break point located at `Line` in `Module`.

ir() -> ok

Deletes all existing break points.

ir(Module) -> ok

Types:

- Module = atom()

Deletes all existing break points in `Module`.

ir(Module,Function,Arity) -> ok | {error, What}

Types:

- Module = atom()
- Function = atom()
- Arity = int()
- What = badarg | function_not_found

Deletes break points at the first line in every clause of the `Module:Function/Arity` function.

`ibd(Module,Line) -> ok | {error, What}`

Types:

- Module = atom()
- Line = int()
- What = badarg | no_break

Makes the break point at `Line` in `Module` inactive. The break point still exists, but no processes will be stopped at the break point.

`ibe(Module,Line) -> ok | {error, What}`

Types:

- Module = atom()
- Line = int()
- What = badarg | no_break

Makes the break point at `Line` in `Module` active. Processes will again be stopped at the break point.

`iba(Module,Line,Action) -> ok | {error, What}`

Types:

- Module = atom()
- Line = int()
- Action = enable | disable | delete
- What = badarg | no_break

Sets the status of the break point at `Line` in `Module` after it is triggered the next time. `Action` is: `enable`, `disable`, or `delete`.

`ibc(Module,Line,Function) -> ok | {error, What}`

Types:

- Module = atom()
- Line = int()
- Function = {M,F}
- Mod = atom()
- Func = atom()
- What = badarg | no_break

Makes the break point at `Line` in `Module` conditional. `Function` is called whenever the break point is reached. `Function` is a tuple {`Mod`,`Func`}. `Function` must have `arity 1` and return either `true` or `false`. This way, the break point either triggers, or not. The argument to `Function` is the current variable bindings of the process at the place of the break point. The bindings can be inspected using `int:get_binding/2`.

`ipb() -> ok`

Makes a printout of all existing break points.

```
ipb(Module) -> ok
```

>   Types:

>   - Module = atom()

>   Makes a printout of all existing break points located in `Module`.

```
iv() -> atom()
```

>   Returns the current version number of the interpreter.

```
help() -> ok
```

>   Prints help text.

Usage

Refer to the Debugger User's Guide for information about the graphical interface.

See Also

int(3), code(3)

```
ipb(Module) -> ok
```

>   Types:

>   - Module = atom()

int

Erlang Module

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

From the shell, it is possible to:

- Specify which modules should 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 via a third process, called the meta process. It is possible, but not recommended, to implement your own attached process. See `int.erl` for available functions and `dbg_ui_trace.erl` for possible messages.

Breakpoints

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

A breakpoint have a status, a trigger action and may have a condition associated with it. The status is either *active* or *inactive*. An inactive breakpoint is ignored. When a breakpoint is reached, the trigger action specifies if the breakpoint should continue to be active (*enable*), if it should become inactive (*disable*), or if it should be removed (*delete*). A condition is a tuple {`Module`,`Name`}. When the breakpoint is reached, `Module:Name(Bindings)` is called. If this evaluates to `true`, execution will stop. If this evaluates to `false`, the breakpoint is ignored. `Bindings` contains the current variable bindings, use `get_binding` to retrieve the value for a given variable.

By default, a breakpoint is active, has trigger action enable and has no condition associated with it. For more detailed information about breakpoints, refer to Debugger User's Guide.

Exports

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

>   Types:
>   - AbsModule = Module | File | [Module | File]
>   - Module = atom()
>   - File = string()
>   - Options = term()
>
>   Interprets the specified module. `i/1,2` interprets the module only at the current node. `ni/1,2` interprets the module at all known nodes.
>
>   A module may be given by its module name (atom) or by its file name. If given 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 a `src` directory next to it.
>
>   If given by its file name, the file name may include a path and the `.erl` extension may 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 needs both the source code and the object code, and the object code *must* include debug information. That is, only modules compiled with the option `debug_info` set can 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 node only at the current node. `nn/1` stops interpreting the module at all known nodes.
>
>   As for `i/1` and `ni/1`, a module may be given by either its module name or its file name.

`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 file name `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
>
>   Checks if a module is possible to interpret. The module can be given by its module name `Module` or its file name `File`.
>
>   The function returns `true` if both source code and object code for the module is found, and the module has been compiled with the option `debug_info` set.
>
>   The function returns {error,Reason} where `Reason` is `no_src` if no source code is found, `no_beam` if no object code is found, `no_debug_info` if the module has not been compiled with the option `debug_info` set, or `badarg` if `AbsModule` does not exist.

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()]
>
>   Get and set when and how to automatically attach to a process executing code in interpreted modules. `false` means never attach, this is the default. Otherwise automatic attach is defined by a list of flags and a function. The following flags may be specified:
>
>   - `init` - attach when a process for the very 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` will be called as:
>
>   `spawn(Module, Name, [Pid | Args])`
>
>   The resulting process can attach to, i.e. send commands to and receive messages from, the interpreted process, see above.

stack_trace() -> Flag
stack_trace(Flag)

>   Types:
>
>   - Flag = all | no_tail | false

Get and set how to save call frames in the stack. This makes it possible to inspect the call chain of a process.

- `all` - save all call frames. This is the default.
- `no_tail` - save all call frames, except for tail recursive calls. This option consumes less memory and may be necessary to use for processes with long lifetimes and many tail recursive calls.
- `false` - do not save call frames.

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

> Types:
> - Module = atom()
> - Line = integer()
>
> Creates a breakpoint at `Line` in `Module`.

`delete_break(Module, Line) -> ok`

> Types:
> - Module = atom()
> - Line = integer()
>
> Deletes the breakpoint located at `Line` in `Module`.

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

> Types:
> - Module = Name = atom()
> - Function = atom()
> - Arity = integer()
>
> Create a breakpoint at the first line of every clause of the `Module:Function/Arity` function.

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

> Types:
> - Module = Name = atom()
> - Arity = integer()
>
> Delete the breakpoints at the first line of every clause of the `Module:Function/Arity` function.

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

> Deletes all breakpoints, or all breakpoints in `Module`.

`disable_break(Module, Line) -> ok`

> Types:
> - Module = atom()
> - Line = integer()

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

`enable_break(Module, Line) -> ok`

Types:
- Module = atom()
- Line = integer()

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

`action_at_break(Module, Line, Action) -> ok`

Types:
- Module = atom()
- Line = integer()
- Action = enable | disable | delete

Set the trigger action of the breakpoint at `Line` in `Module` to `Action`.

`test_at_break(Module, Line, Function) -> ok`

Types:
- Module = atom()
- Line = integer()
- Function = {Module,Name}
- Module = Name = atom()

Set the condition of the breakpoint at `Line` in `Module` to `Function`.

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

Types:
- Var = atom()
- Bindings = term()
- Value = term()

Retrieve the binding of `Variable`. This function is intended to be used by a breakpoint condition function.

`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 = integer()
> - 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` - additional information.
>
> `Status` is one of:
>
> - `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 has been stopped, normally at a breakpoint. `Info={Module,Line}`.
> - `exit` - the process has 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 = integer()
>
> Resume process execution for `Pid`, or for `c:pid(X,Y,Z)`.

List of Figures

Debugger Application (DEBUGGER)

Index of Modules and Functions

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