# Debugger Application (DEBUGGER)

version 1.5

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

# Contents


1 Debugger User's Guide 1

1.1 The Debugger . . . . . 2

Introduction . . . . . 2

Starting the Debugger . . . . . 2

Preparation . . . . . 2

Interpreting . . . . . 3

Break Points . . . . . 4

Configuring Options . . . . . 6

Saving and Loading debugger states . . . . . 7

Debugging . . . . . 7

The Monitor Window (Main Window) . . . . . 7

The View Window . . . . . 11

The Attach Window . . . . . 12

The Interpret Module Dialog . . . . . 17

Save Settings Dialog . . . . . 19

Load Settings Dialog . . . . . 19

Line Break Dialog . . . . . 20

Conditional Break Dialog . . . . . 21

Function Break Dialog . . . . . 21

Debugging Scenarios . . . . . 22

2 Debugger Reference Manual 27

2.1 i (Module) . . . . . 31

2.2 int (Module) . . . . . 37

List of Figures 45

# Chapter 1

# Debugger User's Guide

The *Debugger* application is a graphical tool which can be used for debugging and testing of Erlang programs. Programs can be single stepped, variables can be displayed, breakpoints can be set and removed etc.

# 1.1 The Debugger

## Introduction

The Erlang source code debugger *(Debugger)* provides mechanisms which make it possible to follow and influence the execution of a program in specified modules, as well as examining the process state after an exit.

A source code debugger is mainly used in order to locate errors in code (bugs), but it can also be used as a tool in order to understand and learn about applications written by others.

Since Erlang is a distributed, concurrent language the Erlang Debugger provides mechanisms to attach and interact with several processes at the same time. All processes running code in interpreted modules are monitored and continuous status information can be displayed for these processes.

Processes can be stopped at certain points (break points) in the source code. There are different types of break points, and different actions can be taken on them once they are triggered.

It must always be taken into consideration that stopping one process may influence another process. A process waiting for some response from a stopped process may cause a *time out* and some other action might be taken. Therefore, it may be necessary to set break points for influenced processes, or change the specified *time out* period (perhaps to `infinity`) during interpretation.

After a process has been attached, the current point of the execution state can be examined and altered. Variable bindings are displayed and updated, for inspection of the message queue.

Manipulating a process state by stepping will allow examination and alteration of a process one expression at a time. It is also possible to manipulate the process state of execution during interpretation. By evaluating user provided expressions (where new values can be bound to existing variables), a bug can be bypassed and the execution of the process resumed in order to locate another bug.

## Starting the Debugger

The Debugger can either be started from the Toolbar by clicking on the Debugger icon, or by calling the function `debugger:start()`.

## Preparation

The Erlang Debugger (also referred to as *Debugger*) is implemented by executing the debugged code in an interpreter. This means that compiled Erlang code is not used when debugging. Instead a special preparation of the source code must be made in order to debug it.

> Note:
> The term *interpret* is used in this documentation to indicate that the code has been prepared in the following fashion.

The preparation consists of parsing and annotating the source code, and storing it in the Debugger's internal code database. The code server is also notified so that the module's compiled code is unloaded, and made unavailable.

When the code has been interpreted, the user may want to do some more specific preparation, such as setting break points and configuring options before actually executing the program to be debugged.

## Interpreting

Prior to all debugging, the modules to be debugged must be interpreted.

> Note:
> Not all modules in a running program need to be interpreted, but be aware that you will not receive any debug information or be able to take control of programs when they are running code from an uninterpreted module.

To interpret a module, select *Module->Interpret* in the Monitor Window, and use the dialog to navigate in the file system. To interpret an Erlang file, or all Erlang files in a directory click either the *Choose* or *All* button.

From the *Options*-menu, the user may set some options to control the behavior of the interpretation. See below for details.

Messages and warnings from interpretation of code will be printed in the Erlang Shell.

### Configuring Macros and Include Paths

After all the applications are compiled (using for instance UNIX `make` and `erlc`), various command line options can be used to specify include paths and macro definitions. The Debugger user interface also has functionality for specifying include paths and macro definitions.

To specify the include path for interpretation, select *Options->Include Directories...* in the Interpret dialog. Then select one of the radio buttons *current directory* and *all directories*, depending on the scope you wish your settings to have. Next you need to enter the directory you wish to add to the include path in the *Include directory* field and press the *Add* button.

To specify macro definitions for interpretation, select *Options->Macro Definitions...* in the Interpret dialog. Then select one of the radio buttons *current directory* and *all directories*, depending on the scope you wish your settings to have. Next you need to enter the name and value of the macro you wish to define in the *Macro* and *Value* fields and press the *Add* button.

# Break Points

Once you have interpreted the source modules for debugging, set break points at relevant locations in the source code. Break points can be specified either on a line basis, or on a function basis. Whichever you decide to use, please note that the break points will always be stored on a line basis.

> Note:
> Break points only stops the execution of the affected process not the whole system.

## Setting Break Points

The easiest way to set a break point is to view the source code and locate the line where you want to have the break, and set it. You do this by selecting your module of choice, from the *Module->View* sub menu in the Monitor Window.This will open a View Window for the selected module. Next you locate the line where you want to have the break point, and double-click on that line.

You may also set break points for a specific function by selecting *Breaks->Function Break...* in the View Window. This will open a Function Break dialog where you may specify a function name and arity. This in turn will lead to a line break being set on the first line of each clause for the specified function.

> Note:
> Once an Attach Window is opened you can then set break points as you did in the View Window.

## Deleting Break Points

Break points can be deleted either with the menu selection *Breaks->Delete All*, or by selecting the *Delete* sub menu item for any of the dynamically added break point entries in the *Breaks* menu. A break point can also be deleted (or added) by double clicking on a source code line.

Disabling is a slightly milder version of deleting a break point. A disabled break point will not stop execution. Its entry will remain in the debugger menus, so that there will be an easy way to re-enable it.

## Status and Action of Break Points

A break point can be in one of two states, either `disabled` or `enabled`. A disabled break point never causes processes to stop the execution. However, it can become enabled, and thus set to trigger next time it is reached.

It is possible to specify what status each break point is to have after it has been triggered. This is defined as an action of the break point. The default action is to keep the break point active *(enabled)*. The action can make the break point inactive *(disabled)* or the break point can be removed *(deleted)*.

## Conditional Break Points

The usual way of setting a break point at a specific line can be cumbersome. For example, if you have realized that your program fails only after the 1000:th iteration in a recursive loop, or only at the entry of a function with a specific argument value, you would have use *Step*, *Continue* or *Next* many times before you come to the point where you can study the behavior of the program. To make this task easier, the programmer can set Conditional Break Points.

A Conditional Break Point evaluates a user-defined conditional function prior to stopping. If the function evaluates to true, the Conditional Break Point will stop execution, otherwise the break point will be silently ignored.

To set a Conditional Break Point, first you need to create a function that takes one argument. When the Conditional Break Point is reached, this function will be called with a list of variable bindings as its argument.

Next, to set the Conditional Break Point, select the *Breaks->Conditional Break...* menu item. Make sure that the position of the break point is entered in the upper two boxes of the dialog. In the lower two boxes, enter the module and function name of the user-defined function to be called.

The user defined function can only take one argument, which are the current variable bindings of the current process. The function `int:get_binding/2` is provided in order to examine the binding of a specified variable, which can be used in order to decide whether the Conditional Break Point should trigger or not. `int:get_binding(Variable, Bindings)` returns `unbound` or `{value, Value}`.

In the example below a Conditional Break Point calling `c_test:c_break/1` is added at line 8 in the module `fac`. Each time the break point is reached, the function is called, and when `N` is equal to 3 it returns `true`, and the execution halts at line 8 in module `foo`.

This is the code that is being debugged:

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

Figure 1.1: Conditional Break

This is the function called by the Conditional Break Point:

```
-module (c_test).
-export ([c_break/1]).
c_break (Bindings) ->
    case int:get_binding ('N', Bindings) of
        {value, 3} ->
            true;
        _ ->
            false
    end.
```

### Trigger actions

All break points have associated trigger actions, i.e. a specification of what should happen when a process reaches a break point.

The trigger action is selected from among one of the following three:

***Enable*** If an enabled break point with the trigger action *Enable* is reached, execution will stop at the break point, and the break point will remain as it is.

***Disable*** If an enabled break point with the trigger action *Disable* is reached execution will stop at the break point, and the break point will become disabled.

***Delete*** If an enabled break point with the trigger action *Delete* is reached, execution will stop at the break point, and the break point will be deleted.

> Note:
> The trigger action has no function for relating to disabled break points.

## Configuring Options

The major option to consider when preparing for a debugging session is the *Attach* option. The following options can be chosen if the user wishes a Attach Window to be automatically opened on a particular occasion:

***First call in the process*** An Attach Window is opened at the first function call in the process. Execution is stopped at the first line of the function, and the Debugger expects the user to give a command.

***At break points*** An Attach Window is opened at the first reached break point in the process, unless it has already been opened. Execution is stopped at the break point, and the user is expected to give a command.

***At process termination*** An Attach Window is opened when the process terminates.

## Saving and Loading debugger states

When debugging a large application, the number of modules interpreted, and the number of set break points may be quite large. To save the user from manually having to re-interpret and re-create break points, all this information can be saved as a Debugger state [page 19]. This state may later be retrieved, so one can easily set up the same debugging session again without having to go through the entire preparation process.

> Note:
> Interpretation takes some time to perform, so loading the state where several modules are interpreted may be slow, even when loading a saved Debugger State.

## Debugging

See Debugging Scenarios [page 22] for more detailed instructions on different ways to debug your code.

In order to attach a process, double click the desired process line or select a process with a click and select Attach in the Process menu.

## The Monitor Window (Main Window)

The Monitor Window is the Main Window of the Debugger. Its contents and functionality is described in following subsections.

The Monitor Window can be opened at a node running on a machine which supports the GS graphics libraries, thereby, making it possible to attach processes at nodes where graphics are not available.

Debugger functions can be accessed from the Monitor Window.

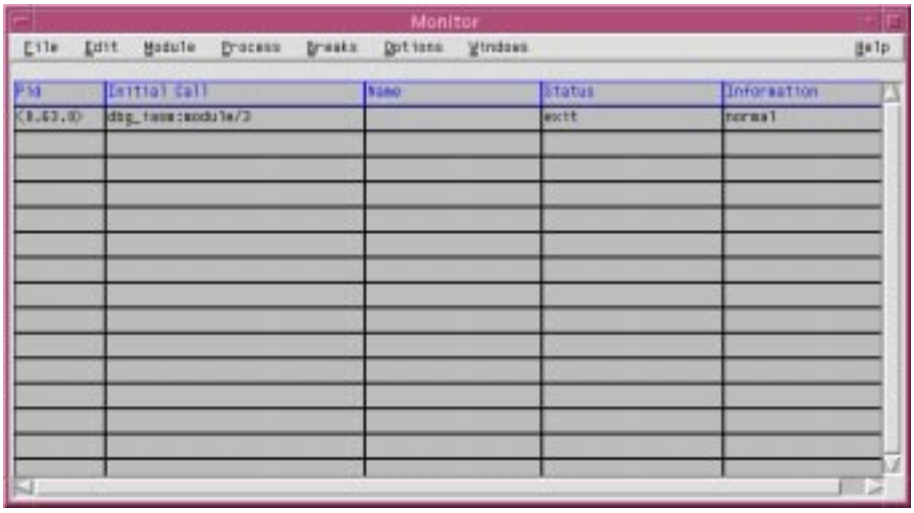


Figure 1.2: The Monitor Window.

## The Process Display

In the Monitor Window, status information is displayed for all debugged processes. There is one process per line, where each line is divided into the following fields:

***Pid*** The process identity of the process displayed in the line.

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

***Name*** If the process is registered, this field shows its registered name.

***Status*** The current state of the displayed process, where the state can be one of the following:

- ***idle*** The interpreted function call has returned a value, and is no longer running interpreted code.
- ***running*** The process is running interpreted code or is waiting for a compiled function call to return a value.
- ***waiting*** The process is waiting for messages at a receive statement, or for a timeout at a receive statement.
- ***break*** The process has stopped execution. The Debugger will expect some user interaction, when a break point is reached or code is single stepped, to continue execution. Additional information shows the location of the current source code.
- ***exit*** The process has terminated.
- ***no_conn*** The connection to the Erlang node holding the process has been broken.

***Information*** Contains either information regarding the location (where the process stopped execution), {`Module, LineNumber`}, or the reason why the process terminated.

## The File menu

***Load Settings*** Opens a dialog for restoring a previously saved debugger state. See the Load Settings Dialog [page 19].

***Save Settings*** Saves the current debugger state in a user specified file. This will save the set of interpreted files, break points, and selected options for future restoration of the debugger state. The default directory is `.erlang_tools/debugger/` created in the user's home directory. See the Save Settings Dialog [page 19]

***Exit*** Terminates the Monitor Window, all current processes and the view and Attach Windows. The user will be queried about whether the Debugger state should be saved. See above.

## The Edit menu

***Clear*** Removes all terminated processes. The Attach Windows of removed processes are also closed.

***Kill All*** Terminates all processes with references to interpreted modules, using the BIF `exit/2` with `kill` as reason.

## The Module menu

This menu contains functions for operating on modules.

***Interpret...*** Opens the Interpret Module dialog [page 17] where new modules to be interpreted can be specified.

***Delete All*** De-interpret all modules. Processes with references to interpreted modules will be killed, with the exception of Idle processes.

***Delete*** De-interprets the selected module. Only processes with references to this module will be killed.

***View*** Opens the View Module Window for the selected module. See the View Window [page 11] section for details.

## The Process menu

This menu has functions that apply to the process in focus.

***Step*** Enables the user to view and execute the interpreted code step by step.
When the next expression contains a `spawn` function call, a new Attach Window will be automatically opened to follow the interpreted code of the new process.

***Next*** Evaluates the expression on the current line, and stop on the next line.

***Continue*** Resumes the execution, continue until the next break point.

***Finish*** Continues execution in the selected process until the current function call returns a value.

***Attach*** Attaches the selected process. An Attach Window [page 12] is opened.

***Kill*** Terminates the selected process, using `exit/2` with reason `kill`. If the process is already terminated this menu item is disabled.

## The Breaks menu

The breaks menu contains functions for breaks, and lists the breaks as they are added or removed, dynamically adding menu items to the breaks menu bar. When creating a break point, you may specify the action to be taken when it is reached by selecting the following options from the menu:

***Line Break...*** Set a break point at a line in a module source file. See also the Line Break Dialog [page 20].

***Conditional Break*** Sets a conditional break point at a line in a module. The name of the module and function which is the condition has to be provided as well. See also the Conditional Break Dialog [page 21].

***Function Break*** Sets a break point at a specified function. This will set a Line Break at the beginning of each clause of the function. See also the Function Break Dialog [page 21].

***Delete All*** Removes all break points.

The rest of the menu contains menu items for all breaks that have been set. Each menu item has the following sub menu items for modifying that specific break point.

***Disable*** Disables the break point. The break point will remain, but nothing will happen when it is reached.

***Delete*** This deletes the break point.

***Trigger Action > Delete*** Specifies that the break point should be deleted when it is reached.

***Trigger Action > Enable*** Specifies that the break point shall remain active when it is reached.

***Trigger Action > Disable*** Specifies that the break point shall be disabled when it is reached.

A line break can also be inserted and removed by double clicking on the specific row in a View or Attach Window.

## The Options menu

The options menu in the Monitor Window displays functions available only to opened Attach Windows.

***Reset Options*** Sets frame, attach and stack options to the default values.

***Button Frame*** Hide/show the Button Area. See also the illustration of the Attach Window [page 12].

***Evaluator Frame*** Hide/show the Evaluator Area. See also the illustration of the Attach Window [page 12].

***Bindings Frame*** Hide/show the Bindings Area. See also the illustration of the Attach Window [page 12].

***Trace Frame*** Hide/show the Trace Area. See also the illustration of the Attach Window.

***Back Trace Size: 100...*** Opens a dialog for setting the number of stack frames to save during evaluation. This determines how many stack frames can be retraced (by using the *Up* and *Down* buttons to observe previous calls and variable bindings), after a break function has been triggered.

***Attach*** This sub menu contains menu items for controlling the automatic opening of Attach Windows.

- ***First Call*** When this check box is selected a new Attach Window will be opened for all new processes running interpreted code.
- ***On Exit*** When this check box is selected a new Attach Window will be opened for processes running interpreted code terminates.
- ***On Break*** When this check box is selected a new Attach Window will be opened when a break point is reached.
- ***All*** This menu item selects all check boxes described above.
- ***Never*** This menu item clears all check boxes described above.

***Stack Options Stack On, Tail*** Instructs the Debugger to collect information about the stack, i.e. the call chain for the functions. This way you can move up and down the stack inspecting variables in the calling environments as well as the current position in the code.

***Stack On, no Tail*** Almost the same as above, but does not keep information about the last function call in the body of a function. If processes, which have very long lifetimes and many tail recursive calls, are interpreted you may run out of memory *unless* this option selected.

***Stack Off*** Keep no call frames in the stack at all.

### The Windows menu

From this menu, all Debugger windows are easily reachable.

### The Help menu

***Help*** Provides help for the Debugger.

## The View Window

The purpose of the View Window is to view source code and change break points.

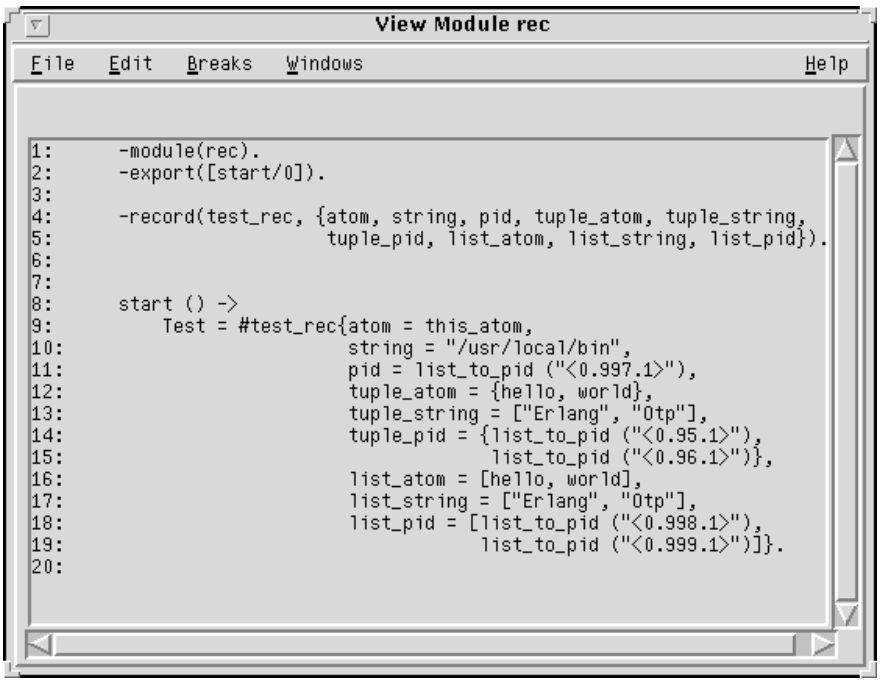


Figure 1.3: The View Window

### Code area

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 break point functions available from the menu. Double clicking a line will set a break point on that line. Double clicking a line with an existing break point will remove the break point.

Note:
*Note:* Breaks are marked with -@-.

### The File menu

***Close*** Closes the window.

### The Edit menu

***Go to line*** Go to that line in the code.

***Search*** Text search.

### The Breaks menu

This menu has the same functions as the Breaks menu in the Main Window. [page 9]

### The Windows menu

This menu has the same function as the Windows menu in the Main Window. [page 11]

### The Help menu

This menu has the same function as the Help menu in the Main Window. [page 11]

## The Attach Window

In Attach Windows the user interacts with a debugged process. It is possible to single step the execution of a process, inspect variable bindings, manipulate break points, etc.

A process is either attached manually from the Monitor Window, or automatically using the automatic attach facility.

With the commands `debugger:quick(Module, Function, [Arg1, Arg2, ...])` or `debugger:q(Module, Function, [Arg1, Arg2, ...])` an Attach Window can be opened quickly without starting the Monitor Window.

Note:
One Attach Window is opened for each attached process.

Figure 1.4: The Attach Window (without the trace area)

The Attach Window is divided into five parts:

***The Source File Area*** Shows the code being executed.

***The Button Area*** Shows the code being executed and supplies the user with buttons for quick access to frequently used functions. The full set of functions are available from the menus.

***The Evaluator Area*** Allows the user to execute commands in the process environment, change bindings, or evaluate expressions.

***The Bindings Area*** Contains a grid with all the bindings.

***The Trace Area*** Shows a trace output of the process.

> **Note:**
> These areas, with the exception of the Source File Area, can be hidden.

### The Source File Area

The source code is displayed with an extra preceding column where the line numbers are shown. The current line is marked –>. An existing break point at a line is marked -@-. In the example below, the execution of the source code is currently at line 37 (before the execution of the `spawn` BIF). The same example also contains a break point at line 35.

```
34:     List = lists:append(L1, L2),
35:-@-            case lists:member(1, List) of
36:           true ->
37:-->          spawn(ex, start,[]);
```

Active break points are shown in red, while the inactive ones are in blue.

### The Button Area

The Button Area contains the most commonly used commands, corresponding to the different menu items under the Process menu.

> **Note:**
> *Note:* The line indicator on the *Break* button changes as a new line is selected in the code area.

### The Evaluator Area

You can enter any Erlang command into the entry box labeled *Evaluator* and the result will be displayed in the view area below it. The commands will be executed in the environment of the attached process.

### The Bindings Area

All bound variables are listed here, showing their names and values.

Long names or values are truncated or shortened, to get the entire value just click in a field in the bindings area to get the name and value printed in the evaluator area.

Records will not be shown as records, but rather as their internal representation, i.e. as tuples.

By double-clicking on a variable in the Bindings Area the *Record Editor* will be opened. The Record Editor is mainly used to re-bind records in a more easier way, but it also works for other Erlang terms.

Figure 1.5: The Record Editor

> **Note:**
> The Record Editor cannot be used to re-bind references, binaries or ports, because GS cannot handle it.

## The Trace Area

If the *Trace Frame* option has been selected, the Trace Frame will continuously show trace information for the debugged process. Function call Information and function call return values will be written, for each interpreted function call, sent and received messages and for each interpreted `receive` statement. Each trace line is annotated with the following tags:

`++(N) L` Indicates a function call, where `N` is the level of recursion and `L` is the corresponding line of the source code of the call.

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

`==> Pid : Msg` Shows that the message `Msg` is sent to process `Pid`.

`<== Msg` Shows the reception of the message `Msg`.

`++(N) receive` Shows that the process is waiting at a `receive statement`.

`++(N) receive with timeout` Shows that the process is waiting at a `receive statement` with a timeout.

## The File menu

***Close*** Close this window and detach from the process.

## The Edit menu

***Go to line*** Go to that line in the code.

***Search*** Text search.

## The Module menu

In the Module menu there is one *View*-item for each interpreted module. Select one of the items to open a View Window for that module.

## The Process menu

***Step*** Executes next program line, step *into* any function calls within the line. If single step is used on a `spawn`, the spawned process is automatically attached, and a new Attach Window is opened for that process.

***Next*** Executes the entire program line indicated by the arrow.

***Continue*** Continues the execution of the process until a break point is reached

***Finish*** Continues execution of the process until the current function call returns a value.

***Time Out*** Simulates a timeout. This does only have effect if a `receive` expression containing an `after` clause is being executed.

***Skip*** Goes to the next line without executing the current line. This function is useful when a bug has been located but evaluation will commence after a correct expression has been evaluated (in the evaluator area), or a variable binding changed. If the last expression of a function is *skipped*, the atom skipped is returned by that function.
*Note:* By using *Skip* you will modify the normal behavior of the code, and that variables (expected by future calls) to be bound, may be unbound, and cause the process to exit.

***Stop*** Stops the execution.

***Up*** Inspects the previous stack frame. The code area will show the location and the bindings area will show the bindings for that stack frame.

***Down*** Inspects the next stack frame. The code area will show the location and the bindings area will show the bindings for that stack frame.

***Where*** Changes the code area to show the current location of the execution.

***Messages*** Inspects the message queue of the process. The queue is printed in the evaluator.

***Kill*** Kills this process, using `exit/2` with reason `kill`.

## The Breaks menu

***Line Break*** Sets a break point at a line in a module source file.

***Conditional Break*** Sets a conditional break point at a line in a module. The name of the module and function (which is the condition) have to be provided as well.

***Function Break*** Sets a break point at a specified function. This will set a Line Break at the beginning of each clause of the function.

***Delete All*** Removes all break points.

The rest of the menu contains menu items for all breaks that have been set. Each menu item has the following sub-menu items for modifying specific break points.

### The Options menu

This menu controls the appearance and behavior of the Attach Window.

See also the Options menu in the Main Window [page 10] that holds functions for setting the default options.

***Show Buttons*** Hide/show the Button Area.

***Show Evaluator*** Hide/show the Evaluator Area.

***Show Bindings*** Hide/show the Bindings Area.

***Show Trace*** Hide/show the Trace Area.

### The Windows menu

***Windows*** The Monitor Window and the View and Attach Windows are managed through a windows manager. The Monitor Window is always on the top followed by the other windows, sorted by their process number.
*Note:* The windows manager is not stable during distribution.

### The Help menu

***Help*** Provides help for the Source Code Debugger.

## The Interpret Module Dialog

The Interpret Module Dialog is used for selecting which files to interpret, and to configure macro definitions and include paths. In the file selection list, only directories and files ending with an `.erl` suffix are displayed. When a file is already interpreted, it is marked with a "*" in front of the name.

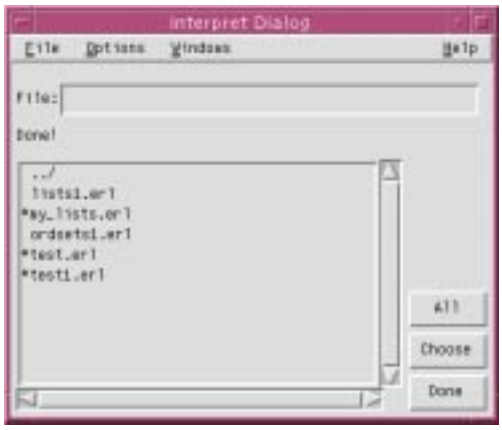


Figure 1.6: The Interpret dialog

To interpret a single file, select it in the list and click *Done*.

To interpret all files in the directory, click *All*.

Compilation/interpretation output is shown in the shell. If a module cannot be interpreted a dialog will pop up.

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

## Macro definitions

To define macros to be used during interpretation, select the menu item *Options->Macro Definitions...*.

This will open a dialog which allows the user to add macro definitions for interpretation.

Macro definitions can be specific to a directory, or valid for any and all directories. If the *current directory* radio button is selected, all definitions added apply only to interpretations made in the current directory in the Interpret Dialog Window. Note that the settings shown in the Macro Definitions change as the user changes directory in the Interpret Dialog.

If the *all directories* radio button is selected, the definitions added apply to interpretations made in all directories.

## Include Directories

To specify which include directories are to be used during interpretation, select the menu item *Options->Include Directories...*.

This will open a dialog which allows the user to add include directories for interpretation.

A set of include directories can be specific to a directory, or valid for any and all directories. If the *current directory* radio button is selected, all include directories added apply only to interpretations made in the current directory in the Interpret Dialog Window. Note that the settings shown in the Include Directories change as the user changes directory in the Interpret Dialog.

If the *all directories* radio button is selected, the include directories added apply to interpretations made in all directories.

## Save Settings Dialog

The Save Settings Dialog is used to save the current state of the Debugger in a file. By doing this, the state can easily be restored at a future time, using the *File->Load Settings* function in the Monitor Window.

Figure 1.7: The Save Settings Dialog

The saved state contains the set of files that have been interpreted, the break points and the debugging options. When the state is restored later on, the set of files will be interpreted again, therefore, a large set of interpreted files will take some time to restore.

The file dialog will automatically be positioned at a configuration directory in the user's home directory. Enter the name of the file where you want the state to be saved. Make sure that the file has `.state` as its extension.

If you wish, you may use the dialog to navigate within the directory structure and place your state file in another directory.

## Load Settings Dialog

The Load Settings Dialog is opened when the user wants to restore a previously saved debugger state. The data that is restored is the set of files that was interpreted, the break points and the debugging options. The files will be interpreted again, so the operation may take some time.

Figure 1.8: The Load Settings Dialog

When opened, the dialog box will show the user's default debugger state directory. Simply select one of the files with a `.state` extension to restore that state.

## Line Break Dialog

This dialog will open when the user requests a break point on a code line to be set.

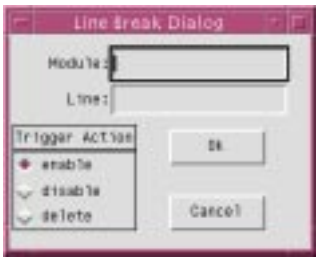


Figure 1.9: The Line Break Dialog

If the dialog is opened from an Attach Window or a View Window, and a line has been selected, the fields will already be filled in with a module and a line number. If the dialog is opened from the Monitor Window, or if a line is not selected, some of the fields may not be filled in.

Make sure that all fields are properly filled in, then select one of the radio buttons and click the *OK* button to set the break point.

For a description of the trigger actions, see the section on Trigger Actions [page 6].

## Conditional Break Dialog

This dialog is opened when the user wants to set a conditional break point at a particular line of source code.

As with the Line Break dialog, if it is opened from an Attach Window or a View Window, and a line has been selected, some of the fields will already be filled in with a module and a line number. If the dialog is opened from the Monitor Window, or if a line is not selected, some of the fields may not be filled in.

For detailed information about how to create and set up conditional break points, please refer to the section Conditional Break Points [page 5].

Figure 1.10: The Conditional Break Dialog

For a description of the trigger actions, see the section on Trigger Actions [page 6].

## Function Break Dialog

This dialog is opened when the user sets a functional break point.

If the dialog is opened from an Attach Window or a View Window, the Module field will already be filled in. If the dialog is opened from the Monitor Window, no fields will be filled.

Enter the module, name and arity of an existing function in the fields. A Line Break Point will be created for the first line of each clause of the specified function.

> **Note:**
> Line Break Points are not associated with the function name once they have been created and therefore *must be* removed/changed individually .

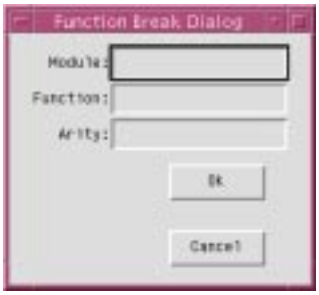


Figure 1.11: The Function Break Dialog

For a description of the trigger actions, see the section on Trigger Actions [page 6].

## Debugging Scenarios

When one is faced with a particular problem, there are many ways of finding a solution. However, there is often a particularly suitable way of proceeding that makes problem solving easier.

Some the more common problems faced are:

- Where did it crash/stop? See the Trap the exit [page 22] section.
- Why did it not start? See the Step from start [page 25] section.
- Automating with Conditional Break Points. See the Break When Needed [page 25] section.
- Debugging remote nodes. See the Debugging remote nodes [page 25] section.
- Debugging non-graphical nodes. See the Debugging non-graphical nodes [page 26] section.

In this section we describe some typical problem scenarios, and some ways of using the Debugger in a way that has proven very useful.

### Scenario 1: Trap the exit

**Problem** Sometimes a program consisting of several processes crashes or blocks. No matter how simple the reason for this may be, it may still be difficult to find.

Often the cause is that some process in the program fails to start up properly. To see if this is the case, simply set the debugger to display a window for all exited processes by following the instructions below.

**Solution** Interpret all modules of the program, and then select the menu item *Options->Attach->On Exit*. This will cause a window to be opened for all processes that terminate. In this way you will find if any of your processes have terminated unexpectedly. An Attach Window will open showing the exact location.

Here you may also use the *Up* and *Down* buttons to trace back (or forward) through the execution to previous stack frames to observe prior calls and variable bindings.

**Example** In the program below, there are two minor errors. This example shows how to use the Debugger to find them.

```
-module(scenario1).
-author('olin@smaug').

%%-compile(export_all).
-export([test/0, ping/1, quit/1]).
%% After first failure, uncomment this:
 -export([init/0]).


test() ->
    Pid = spawn(?MODULE, init,[]),
    io:format("The Pid is: ~p~n",[Pid]),
    Pid.

ping(Pid) ->
    Pid ! {self(), ping},
    receive
        pong ->
            ok
    after
        2000 ->
            dead
    end.

quit(Pid) ->
    Pid ! quit.



init()  ->
    io:format("Initializing...~n"),
    loop().



loop() ->
    receive
        {Pid, ping} ->
            Pid ! pong
            %% After 2nd failure, uncomment this:
            %%,loop()
                ;
        quit ->
            ok
    after
        10000 ->
            io:format("Timeout!~n"),
            loop()
    end.
```

The purpose of the program is to start a server process (using `test/0`) that can be tested with the `ping/1` function.

If no errors existed the program would give the following result:

```
Eshell V4.7.1  (abort with ^G)
1> Pid = scenario1:test().
The Pid is: <0.30.0>
Initializing...
<0.30.0>
2> scenario1:ping(Pid).
ok
3> scenario1:ping(Pid).
ok
4> scenario1:ping(Pid).
ok
5> scenario1:quit(Pid).
quit
```

If we compile and try to start the program from an Erlang Shell, and then call the `ping/1` function, we will not get any response from the server. See the following transcript from the Erlang Shell:

```
Eshell V4.7.1  (abort with ^G)
1> c(scenario1).
./scenario1.erl:54: Warning: function init/0 not called
{ok,scenario1}
2> Pid = scenario1:test().
The Pid is: <0.34.0>
<0.34.0>
3> scenario1:ping(Pid).
dead
4>
```

Apparently the server is not responding. Obviously, the compilation warning gives some clue to what the problem is, but for this example we will disregard this.

Instead we use the Debugger to try to locate the problem. Interpret the module `scenario1` and set the options so that an Attach Window is opened when a process exits.

Again attempt to start the server. This time we will get an error report in the Erlang Shell, and in addition to this an Attach Window will be opened.

```
5> Pid2 = scenario1:test().
The Pid is: <0.54.0>
<0.54.0>
6>
=ERROR REPORT==== 15-Jun-1998::15:58:20 ===
** undefined function: scenario1:init[] **
```

From both the Attach Window and the error report in the Erlang Shell one can immediately see that the function `scenario1:init/0` is not defined. The `-export()` directive for it has simply been forgotten.

To remedy this, uncomment the line...

```
%%-export([init/0]).
```

...at the beginning of the file `scenario1.erl`.

Compile or interpret the module `scenario1.erl` again, and try it.

This time, if we start the program, the server will be started and respond to the first `ping/1` call. But at the second call to `ping/1` it will not respond. (See the transcript below).

```
2> scenario1:test().
The Pid is: <0.34.0>
Initializing...
<0.34.0>
3> scenario1:ping(pid(0,34,0)).
ok
4> scenario1:ping(pid(0,34,0)).
dead
```

If we are still using the Debugger, an Attach Window will automatically be opened in connection with the first call to `ping/1`. In that window we will see that the process has exited with the reason `normal`.

After some inspection of the source code we find that the case of the receive statement that handles the ping message did not call the loop function recursively. Therefore, it runs out of code to execute, which results in termination with the reason `normal`.

### Scenario 2: Step from start

**Problem** Sometimes your program does not seem to start up properly, and you have no clue where the problem lies.

**Solution** Go through the program step-by-step. To do this, interpret all modules, then select the menu item *Options->Attach->First Call*. Then use the *Step* and *Next* buttons to go through the execution step-by-step.

### Scenario 3: Break when needed

See the Conditional Break Points [page 5] section for details on setting a conditional break point that postpones activation till a specified limit has been reached.

### Scenario 4: Debugging Remote Nodes

**Problem** There are few problems with debugging remote nodes, since the Debugger automatically supports debugging of distributed programs.

**Solution** When the debugger is running on a distributed Erlang system, any modules that are interpreted become interpreted on all known Erlang nodes in the network. Thus any processes running interpreted code on any node will be debugged, and depending on the attach options, an Attach Window will be opened.

Any break points set are set on all nodes, making it impossible to have different break points on different nodes.

Essentially, there is no difference between debugging processes on the local node where the Debugger is running, and debugging processes on remote nodes.

## Scenario 5: Debugging non-graphical nodes

**Problem** Using the graphical Debugger on a system that has no graphical display possibilities.

**Solution** To debug for instance an embedded system that has no graphical display possibilities, make sure that the system is running as a node in a distributed Erlang system. The node be started using the `-name` or `-sname` options to the `erl` program. We will refer to the system without graphical capabilities as the *embedded node*.

Start an Erlang node on a system that has graphical capabilities (using the `-name` or `-sname` options), and make sure that the two nodes are aware of each other (for instance by using `net_adm:ping/1`). We will refer to the node with graphical capabilities as the *Debugger node*.

Locate the source code of the embedded system and Interpret the code. The interpretation will be propagated to the embedded node (and any other known nodes).

After this set up, you can continue debugging using any of the other methods described in this document. See the section Debugging Remote Nodes [page 25].

# Debugger Reference Manual

## Short Summaries

- Erlang Module i [page 31] – Interpreter (debugger) Interface.
- Erlang Module int [page 37] – Interpreter (Debugger) Interface.

## i

The following functions are exported:

- `im() -> pid()`
  [page 31]
- `ii(AbsModule) -> {module, Module} | error`
  [page 31] Interprets a module
- `iq(Module) -> ok`
  [page 32] Do not interpret a module
- `ini(Module) -> {module,Module} | error | ok`
  [page 32]
- `inq(Module) -> ok`
  [page 32]
- `il() -> ok`
  [page 32]
- `ip() -> ok`
  [page 32]
- `ic() -> ok`
  [page 32]
- `iaa(Flag) -> true`
  [page 32] Automatically attaches to a process
- `iaa(Flag,Function) -> true`
  [page 32] Automatically attach to a process using `Function` as the attachment method.
- `ist(Flag) -> true`
  [page 33] Sets the usage of call frame inspections
- `ia(Pid) -> ok | no_proc`
  [page 33] Attaches to a process

- `ia(X,Y,Z) -> ok | no_proc`
  [page 33] Attaches to a process
- `ia(Pid,Function) -> ok | no_proc`
  [page 33] Attaches to a process
- `ia(X,Y,Z,Function) -> ok | no_proc`
  [page 33] Attaches to a process
- `ib(Module,Line) -> ok | {error, What}`
  [page 34] Sets a break point
- `ib(Module,Function,Arity) -> ok | {error, What}`
  [page 34] Sets a break point in a function
- `ir(Module,Line) -> ok | {error, What}`
  [page 34] Deletes a break point
- `ir() -> ok`
  [page 34]
- `ir(Module) -> ok`
  [page 34] Deletes all break points in a module
- `ir(Module,Function,Arity) -> ok | {error, What}`
  [page 34] Deletes a break point in a function
- `ibd(Module,Line) -> ok | {error, What}`
  [page 35] Disables a break point
- `ibe(Module,Line) -> ok | {error, What}`
  [page 35] Enables a break point
- `iba(Module,Line,Action) -> ok | {error, What}`
  [page 35] Sets a break point trigger
- `ibc(Module,Line,Function) -> ok | {error, What}`
  [page 35] Sets a conditional test at a break point
- `ipb() -> ok`
  [page 35]
- `ipb(Module) -> ok`
  [page 36] Makes a printout of all break points in module
- `iv() -> atom()`
  [page 36]
- `help() -> ok`
  [page 36]

## int

The following functions are exported:

- `m() -> pid()`
  [page 37]
- `i(AbsModule) -> {module, Module} | error`
  [page 37] Interprets a module
- `i(AbsModule,Options) -> {module, Module} | error`
  [page 38] Interprets a module

- `a(AbsModule) -> {module, Module} | error`
  [page 38] Interprets a module
- `a(AbsModule,Options) -> {module, Module} | error`
  [page 38] Interprets a module
- `n(Module) -> ok`
  [page 39] Do not interpret a module
- `ni(AbsModule) -> {module,Module} | error | ok`
  [page 39]
- `ni(AbsModule,Options) -> {module,Module} | error | ok`
  [page 39]
- `na(AbsModule) -> {module,Module} | error | ok`
  [page 39]
- `na(AbsModule,Options) -> {module,Module} | error | ok`
  [page 39]
- `nn(Module) -> ok`
  [page 39]
- `interpreted() -> [Module]`
  [page 39] Returns a list of which modules are interpreted
- `version() -> atom()`
  [page 39]
- `auto_attach(Flag) -> true`
  [page 39] Automatically attaches to a process
- `auto_attach(Flag,Function) -> true`
  [page 40] Automatically attach to a process using `Function` as the attachment method
- `stack_trace(Flag) -> true`
  [page 40] Sets the usage of call frame inspections
- `snapshot() -> [Snap]`
  [page 40] Gets the current status of all interpreted processes
- `continue(Pid) -> ok | {error, not_interpreted}`
  [page 41]
- `continue(X,Y,Z) -> ok | {error, not_interpreted}`
  [page 41]
- `clear() -> ok`
  [page 41]
- `file(Module) -> FileName | {error, not_loaded}`
  [page 41] Gets the name of the loaded source code
- `break(Module,Line) -> ok | {error, What}`
  [page 41] Sets a break point
- `delete_break(Module,Line) -> ok | {error, What}`
  [page 41] Deletes a break point
- `no_break() -> ok`
  [page 41]
- `no_break(Module) -> ok`
  [page 41] Deletes all break points in a module
- `break_in(Module,Function,Arity) -> ok | {error, What}`
  [page 41] Sets a break point in a function

- `del_break_in(Module,Function,Arity) -> ok | {error, What}`
  [page 42] Deletes a break point in a function
- `disable_break(Module,Line) -> ok | {error, What}`
  [page 42] Disables a break point
- `enable_break(Module,Line) -> ok | {error, What}`
  [page 42] Enables a break point
- `action_at_break(Module,Line,Action) -> ok | {error, What}`
  [page 42] Sets a break point trigger
- `test_at_break(Module,Line,Function) -> ok | {error, What}`
  [page 42] Sets a conditional test at a break point
- `get_binding(Variable,Bs) -> {value, Value} | unbound`
  [page 43] Gets a variable binding
- `all_breaks() -> [{Break, Options}]`
  [page 43] Gets all break points
- `all_breaks(Module) -> [{Break, Options}]`
  [page 43] Gets all existing break points in a module

# i (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)

# int (Module)

The module `int` provides an interface for the Erlang interpreter (debugger). The graphical interface can be opened, but there are also commands available for interacting with the interpreter from the Erlang shell.

The purpose of the interpreter is to provide mechanisms which makes it possible to monitor what is going on while processes execute specified modules, or when processes crash.

The following features are provided to assist the user to catch bugs:

```
o Specify which module(s) to be interpreted.

o Make processes stop on specified break points.

o Examine what has happened when a process has
  stopped, or crashed, by means of process attachment.
  This includes inspecting variable bindings.

o Change processes, so that you can experiment
  by correcting the effects of one bug and proceed to
  the next one.

o Single step the execution.

o Monitor the current status of all interpreted
  processes. Processes spread over several Erlang
  nodes can all be monitored and attached.
```

If a network of Erlang nodes, break points are always updated on all nodes.

## Exports

`m() -> pid()`

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

`i(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.

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

Types:

- AbsModule = atom() | string() | [atom() | string()]
- Module = atom()
- Options = [opt()]
- opt() = verbose | {i,dir()} | {d,macro()} | {d,macro(),term()}
- dir() = string()
- macro() = atom()

As above, but verbose information is given in `Options` is one of:

`verbose` Print verbose information.

`{i,Dir}` Add `Dir` to the list of directories to be searched when including a file.

`{d,Macro}`

`{d,Macro,Value}` Defines a macro `Macro` to have the value `Value`. The default is `true`).

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

Types:

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

*Obsolete* function. It has the same functionality as `int:i/1`.

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

Types:

- AbsModule = atom() | string() | [atom() | string()]
- Module = atom()
- Options = [opt()]

- opt() = verbose | {i,dir()} | {d,macro()} | {d,macro(),term()}
- dir() = string()
- macro() = atom()

*Obsolete* function. It has the same functionality as `int:i/2`.

`n(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.

```
ni(AbsModule) -> {module,Module} | error | ok
ni(AbsModule,Options) -> {module,Module} | error | ok
na(AbsModule) -> {module,Module} | error | ok
na(AbsModule,Options) -> {module,Module} | error | ok
nn(Module) -> ok
```

Behaves as the corresponding i/1,i/2,a/1,a/2,n/1 functions described above, but on all nodes in the network. These functions always return `ok` if we are `alive`, otherwise as above.

`interpreted() -> [Module]`

Types:

- Module = atom()

Returns a list of all modules currently being interpreted.

`version() -> atom()`

Returns the current version number of the interpreter.

`auto_attach(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 `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.

`auto_attach(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.

`stack_trace(Flag) -> true`

Types:

- Flag = all | true | no_tail | false

The interpreter can keep call frames in the stack for future inspections. Typically, it is possible to 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 a 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.

`snapshot() -> [Snap]`

Types:

- Snap = {Pid, InitialFunc, Status, Info}
- Pid = pid()
- InitialFunc = atom()
- Status = idle | running | waiting | break | exit | no_conn
- Info = {} | {Module, Line} | ExitReason
- Module = atom()
- Line = int()
- ExitReason = term()

Returns a list which contains the current status information of all interpreted processes, `[{Pid,InitialFunc,Status,Info}, ...]` where:

- `Pid` is the process identity of the interpreted process.
- `InitialFunc` is the name of the first interpreted function.
- `Status` is the current status of the process.
- `Info` is additional information if `Status` is break (the tuple {`Module,Line`}) or exit (the exit reason).

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

Order `Pid` to resume the execution.

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

Types:

- X = Y = Z = int()

Order the process c:pid(`X`,`Y`,`Z`) to resume the execution.

`clear() -> ok`

Delete (clear) information for all terminated processes from the interpreter.

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

Types:

- Module = atom()
- FileName = string()

Returns the name of the corresponding source code file last loaded for `Module`. Returns the name with the absolute path of the file.

`break(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 is stopped before the expression at `Line` in `Module` is executed.

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

Types:

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

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

`no_break() -> ok`

Deletes all existing break points.

`no_break(Module) -> ok`

Types:

- Module = atom()

Deletes all existing break points in `Module`.

`break_in(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.

`del_break_in(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.

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

Types:

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

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

`enable_break(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.

`action_at_break(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`.

`test_at_break(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`.

`get_binding(Variable,Bs) -> {value, Value} | unbound`

Types:

- Variable = atom()
- Bs = term()
- Value = term()

Gets the binding of `Variable` in the binding structure `Bs`. `Variable` must be an atom, for example `'Num'`. This function is be used from inside a conditional break point function. `Bs` is supplied as an argument to the conditional test function above.

`all_breaks() -> [{Break, Options}]`

Types:

- Break = {Module, Line}
- Module = atom()
- Line = int()
- Options = term()

Returns a list of all existing break points.

`all_breaks(Module) -> [{Break, Options}]`

Types:

- Module = atom()
- Break = {Module, Line}
- Line = int()
- Options = term()

Returns a list of all existing break points located in `Module`.

## Usage

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

## See Also

i(3), c(3), code(3), error_handler(3)

# List of Figures

## Chapter 1: Debugger User's Guide


1.1 Conditional Break . . . . . 5
1.2 The Monitor Window. . . . . 8
1.3 The View Window . . . . . 11
1.4 The Attach Window (without the trace area) . . . . . 13
1.5 The Record Editor . . . . . 15
1.6 The Interpret dialog . . . . . 17
1.7 The Save Settings Dialog . . . . . 19
1.8 The Load Settings Dialog . . . . . 20
1.9 The Line Break Dialog . . . . . 20
1.10 The Conditional Break Dialog . . . . . 21
1.11 The Function Break Dialog . . . . . 22

# Index

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


`a/1`
*int* , 38

`a/2`
*int* , 38

`action_at_break/3`
*int* , 42

`all_breaks/0`
*int* , 43

`all_breaks/1`
*int* , 43

`auto_attach/1`
*int* , 39

`auto_attach/2`
*int* , 40

`break/2`
*int* , 41

`break_in/3`
*int* , 41

`clear/0`
*int* , 41

`continue/1`
*int* , 41

`continue/3`
*int* , 41

`del_break_in/3`
*int* , 42

`delete_break/2`
*int* , 41

`disable_break/2`
*int* , 42

`enable_break/2`
*int* , 42

`file/1`
*int* , 41

`get_binding/2`
*int* , 43

`help/0`
*i* , 36

*i*
`help/0`, 36
`ia/1`, 33
`ia/2`, 33
`ia/3`, 33
`ia/4`, 33
`iaa/1`, 32
`iaa/2`, 32
`ib/2`, 34
`ib/3`, 34
`iba/3`, 35
`ibc/3`, 35
`ibd/2`, 35
`ibe/2`, 35
`ic/0`, 32
`ii/1`, 31
`il/0`, 32
`im/0`, 31
`ini/1`, 32
`inq/1`, 32
`ip/0`, 32
`ipb/0`, 35
`ipb/1`, 36
`iq/1`, 32
`ir/0`, 34
`ir/1`, 34
`ir/2`, 34
`ir/3`, 34
`ist/1`, 33

iv/0, 36

i/1
*int* , 37

i/2
*int* , 38

ia/1
*i* , 33

ia/2
*i* , 33

ia/3
*i* , 33

ia/4
*i* , 33

iaa/1
*i* , 32

iaa/2
*i* , 32

ib/2
*i* , 34

ib/3
*i* , 34

iba/3
*i* , 35

ibc/3
*i* , 35

ibd/2
*i* , 35

ibe/2
*i* , 35

ic/0
*i* , 32

ii/1
*i* , 31

il/0
*i* , 32

im/0
*i* , 31

ini/1
*i* , 32

inq/1
*i* , 32

*int*
a/1, 38
a/2, 38
action_at_break/3, 42
all_breaks/0, 43
all_breaks/1, 43
auto_attach/1, 39
auto_attach/2, 40
break/2, 41
break_in/3, 41
clear/0, 41
continue/1, 41
continue/3, 41
del_break_in/3, 42
delete_break/2, 41
disable_break/2, 42
enable_break/2, 42
file/1, 41
get_binding/2, 43
i/1, 37
i/2, 38
interpreted/0, 39
m/0, 37
n/1, 39
na/1, 39
na/2, 39
ni/1, 39
ni/2, 39
nn/1, 39
no_break/0, 41
no_break/1, 41
snapshot/0, 40
stack_trace/1, 40
test_at_break/3, 42
version/0, 39

interpreted/0
*int* , 39

ip/0
*i* , 32

ipb/0
*i* , 35

ipb/1
*i* , 36

iq/1
*i* , 32

ir/0
*i* , 34

ir/1
*i* , 34

ir/2
*i* , 34

`ir/3`
*i* , 34

`ist/1`
*i* , 33

`iv/0`
*i* , 36

`m/0`
*int* , 37

`n/1`
*int* , 39

`na/1`
*int* , 39

`na/2`
*int* , 39

`ni/1`
*int* , 39

`ni/2`
*int* , 39

`nn/1`
*int* , 39

`no_break/0`
*int* , 41

`no_break/1`
*int* , 41

`snapshot/0`
*int* , 40

`stack_trace/1`
*int* , 40

`test_at_break/3`
*int* , 42

`version/0`
*int* , 39