

# Package ‘debug’

November 13, 2009

**Version** 1.2.1

**Author** Mark V. Bravington <mark.bravington@csiro.au>

**Maintainer** Mark V. Bravington <mark.bravington@csiro.au>

**Depends** R (>= 2.9), mvbutils (>= 2.3), tcltk

**Date** 6/11/2010

**Title** MVB’s debugger for R

**Description** Debugger for R functions, with code display, graceful error recovery, line-numbered conditional breakpoints, access to exit code, flow control, and full keyboard input.

**LazyLoad** no

**License** GPL (>= 2)

**ChangeLog** inst/changes.txt

**Repository** CRAN

**Date/Publication** 2009-11-13 10:42:57

## R topics documented:

debug-package . . . . .	2
bp . . . . .	5
fun.locator . . . . .	7
get.retval . . . . .	8
go-skip-qqq . . . . .	9
is.mtraced . . . . .	10
mtrace . . . . .	10
<b>Index</b>	<b>13</b>

**Description**

`debug` is an alternative to `trace` and `browser`, offering:

- a visible code window with line-numbered code and highlighted execution point;
- the ability to set (conditional) breakpoints in advance, at any line number;
- the opportunity to keep going after errors;
- multiple debugging windows open at once (when one debuggee calls another, or itself);
- full debugging of `on.exit` code;
- the ability to move the execution point around without executing intervening statements;
- direct interpretation of typed-in statements, as if they were in the function itself.

Even if you don't write functions, or even if you don't write buggy functions, you may find it helpful to run the debugger on functions in `package:base` or other packages. Watching what actually happens while a function is executing, can be much more informative than staring at a piece of code or terse documentation.

Debugging your function `f` is a two-stage process. First, call `mtrace(f)` to store debugging information on `f`, and to overwrite `f` with a debug-ready version that will call the debugger itself. Second, do whatever you normally do at the command prompt to invoke `f`. This is often a direct call to `f`, but can be any command that eventually results in `f` being invoked. [The third, fourth, etc. stages, in which you actually fix the problem, are not covered here!]

When `f` is invoked, a window will appear at the bottom of the screen, showing the code of `f` with a highlight on the first numbered line. (There is also an asterisk in the far left hand column of the same row, showing that there's a breakpoint.) The command prompt in R will change to `"D(. . )>"`, showing that you are "inside" a function that is being debugged. The debugger is now in "step mode". Anything you type will be evaluated in the frame of the function— this includes assignments and creation of new variables. If you just type `<ENTER>`, the highlighted statement in `f` will be executed. The result of the statement will be printed in the R command window, and the highlight will (probably) move in the `f` code window.

To progress through the code of `f`, you can keep pressing `<ENTER>`, but you can also type `go()` to put the debugger into "go mode", whereupon it will keep executing code statements without manual intervention. In "go mode", nothing will be printed in the command window (except if there are `cat` or `print` calls in the function code) until either:

- the function completes normally, or
- an error occurs in the function, or
- there's a user interrupt (e.g. ESCAPE is pressed), or
- a breakpoint is triggered.

In the first case, control is returned to the normal R command prompt, just as if the debugger had not been used. In the other cases, the `D(...)>` prompt will return and the line associated with the error / interrupt / breakpoint will be highlighted in the code window. You are then back in step mode. If there was an error, you can type statement(s) that will cause the error not to happen when the highlighted line executes again, or you can move the highlighted execution point to another line number by calling `skip`. Execution carries on quite normally after errors, just as if the offending statement had been wrapped in a `try` call. If your function eventually exits normally (i.e. not via `qqq()`, as described next), it will be as if the error never happened (though the error message(s) will be displayed when the R command prompt returns).

When in step mode, you can finish debugging and return to the normal R command prompt by typing `qqq()`. If you type `<ESC>` while in go mode, you should be returned to step mode, but sometimes you may be dumped back at the R command prompt (this is on to-be-fixed list), and sometimes there will be no immediate effect (e.g. if C code is running).

Breakpoints, including conditional breakpoints, are set and cleared by `bp`. Calling `go(n)` puts the debugger into go mode, but also sets a temporary breakpoint at line `n`, which will be triggered the first time execution reaches line `n` but not subsequently.

When the main function code has completed, the debugger moves into any `on.exit` code, which is also displayed and line-numbered in the code window. (Even if there are no calls to `on.exit`, a numbered NULL statement is placed in the exit code section, so that you can always set a "run-until-function-finishes" breakpoint.) If you exit via `qqq()`, the exit code will not be executed first; this can lead to subsequent trouble with open connections, screwed-up `par` values in graphics, etc.. To make sure the exit code does get executed:

- use `skip` to move to the start of the exit code;
- then use `go(n)` to run to the final NULL in the exit code;
- then use `qqq()` to finish debugging.

When you want to restore `f` to its normal non-debugging state (and you are back at the real R command prompt), type `mtrace(f, FALSE)`. To restore all debuggees, type `mtrace.off()` (`qv`). It is advisable not to save functions in an `mtraced` state; to avoid manual untracing and retracing, look up `Save` in package `mvbutils`.

You can debug several functions "at once" (e.g. if `f` calls `g`, you can `mtrace` both `f` and `g`, with `mtrace(g)` called either inside or outside the debugger), causing several code windows to be open simultaneously. If `f` is called again inside `f` (either via some statement in `f`, or from something you type in step mode), another `f` code window will open. The number in the window title is the frame number, and the currently-active frame number is shown in the `D(...)>` prompt.

For statements typed in at the `D(...)>` prompt, only the first syntactically-complete R expression will be executed; thus, typing `a <- 1; a <- 2` will set `a` to 1, but typing `{ a <- 1; a <- 2 }` will set `a` to 2.

S3 methods work fine with `mtrace`, but S4 methods don't unless you do something special. My suspicion is that they will work OK if you can somehow pass the true name of the method to `mtrace`, along with its environment using the `from` argument. This is probably fixable, but low priority unless I get lots of requests; I have no plans to write S4 methods, and hope not to have to debug other people's!

For further information, see R-news 3/3.

## Options

As of version 1.2.0, output is sent *by default* to "stderr"; this means that you get to see the step-mode output even if the debuggee is redirecting "real" output (e.g. via `sink`). If you don't like this (and I believe Tinn-R doesn't), set `options( debug.catfile="stdout" )`.

Command recall is ON by default, but this means that anything typed while debugging will also be seen in `history()` after leaving the debugger. If this is a problem, set `options( debug.command.recall=FALSE )` (probably in `.First`).

In step mode, only objects with `object.size < 8192` bytes will be printed in full by default; for larger objects, a summary is given instead. You can force printing of any individual object via `print`, but you can also increase (or decrease) the threshold to `X` bytes, by setting `options( threshold.debug.autoprint.size=X )`.

Various TCL/TK-related aspects of the code window can be altered:

- `tab.width` defaults to 4, for indenting code lines (not related to TCL/TK)
- `debug.font` defaults to "Courier"; try e.g. `"Courier 24 italic"`
- `debug.height` (in lines) defaults to 10
- `debug.width` (in characters) defaults to 120
- `debug.screen.pos` defaults to "+5-5" for BL corner; try "-5-5" for BR, "-5+5" for TR, "+5+5" for TL.
- `debug.fg` is foreground text colour, defaulting to `black`

## Note

**Bug:** with R 2.10, the return value of `for`- and `while`-loops should be `NULL`, but `debug` is returning the last value computed within the loop.

**Display bugs:** There have been sporadic and unreproducible display problems with the TCL/TK window; sometimes the window frame will appear, but with garbled contents or no contents at all. With `RTERM` in MS-Windows, a couple of `ALT-TABS` and mouse clicks to change focus are sometimes necessary. In extremis, the window will eventually sort itself out if you manually minimize, maximize, and restore it—admittedly an irritation. Although the problem seems less frequent these days, I'm not convinced it is fully solved.

If you encounter this problem, try setting `options( shakeup.debug.windows=TRUE )` (also in `.First`). The price paid for setting this option, is that window focus will switch away from R to the new code window when the latter is first created, which is mildly irritating. On MS-Windows, I have a fix for that too, but it requires a DLL written in Delphi not C, so I can't CRAN it; let me know if you are desperate.

Apparently the **RGtk2** package doesn't play nicely with `debug`—the debugging window displays a blank. I haven't checked this out, but would be grateful for advice.

For ESS users: I'm not an Emacs user and so haven't tried ESS with the **debug** package myself, but a read-through of the ESS documentation suggests that at least one ESS variable may need changing to get the two working optimally, as shown below. Please check the ESS documentation for further details on these points (and see also `README.mvbutils` in the **mvbutils** package). I will update this helpfile when I have more info on what works.

- The prompt will change when the debugger is running, so you may want to change "inferior-ess-prompt". Prompts will always be of the form D(XXX)> where XXX is a positive integer.
- Command recall probably won't work inside the debugger (or if it does, it's thanks to ESS rather than to R's own command recall mechanism). It should be disabled by default; if you do get error messages about "history not available", make sure to set `options( debug.command.recall=FALSE)` before debugging.

### Author(s)

Mark Bravington

### See Also

`mtrace`, `go`, `skip`, `qqq`, `bp`, `get.retval`, `mtrace.off`, `check.for.tracees`

---

bp

*Breakpoints for debugging*

---

### Description

Sets/clears breakpoints (including conditional) breakpoints in functions that have been `mtraced` for debugging.

### Usage

```
bp( line.no, expr=TRUE, fname) # fname rarely needed
```

### Arguments

<code>line.no</code>	line number
<code>expr</code>	unquoted expression to be tested when execution reaches <code>line.no</code>
<code>fname</code>	name of function to twiddle breakpoints in

### Details

Breakpoints can only be set after `mtrace`ing a function, and are normally set while the function is actually being debugged. The simplest way is to look at the code window to identify which lines to stop at, call `bp(n)` for each of those lines, and then call `go()` to enter go mode. Your function code will then be executed without pausing for input until a breakpoint is triggered (or an error occurs, or the function finishes normally). To clear a breakpoint for line `n`, type `bp(n, FALSE)`.

All line-numbered statements actually have an associated breakpoint expression. When the debugger reaches a line-numbered statement, it evaluates the corresponding breakpoint expression in the function's frame. If the result is not identical to `FALSE`, the breakpoint is triggered. By default, all statements have their breakpoint expressions set to `FALSE` (by `mtrace`), except for line 1 where the expression is set to `TRUE`.

After setting a breakpoint for line *n*, you will see an asterisk (\*) in line *n* of the left-hand column of the code window. The asterisk is shown whenever the unevaluated breakpoint expression is not identical to `FALSE`.

Conditional breakpoints are just expressions other than `TRUE` or `FALSE`. To get the debugger to stop at line 5 whenever *a* is greater than *b*, type `bp( 5, a>b)`—don't quote() the breakpoint expression. Any statement, including a braced statement, can be used, and the debugger will only pause if the result is not `FALSE`. You can therefore use "non-breaking breakpoints" to patch expressions into the code. For instance, if you realize that you should have inserted the statement `a <- a+1` just before line 7 of your code, you can type `bp( 7, { a <- a+1; FALSE})`; when the debugger reaches line 7, it will increment *a* but will not switch to step mode, because the overall result was `FALSE`.

Sometimes it is useful to clear the line 1 breakpoint before invoking a function, especially if the function is being called repeatedly. The debugger actually starts in go mode, and does not display a code window or pause for input until a breakpoint is triggered; so if the line 1 breakpoint is cleared, execution can continue at full speed until an error occurs (or another breakpoint is triggered). To adjust breakpoints before a function is invoked, you will need to use the `fname` argument. To set/clear breakpoints in function *f* at lines other than 1, first type `tracees$f$line.list` to see which line numbers correspond to which statements.

Breakpoints in body code apply "globally" to all incarnations of a function, and will be retained when the debugger finishes and the R prompt returns. Breakpoint expressions for *f* will be saved in `tracees$f$breakpoints`.

Breakpoints can be set in `on.exit` code as well (but are specific to the incarnation they are set in). It is often useful to set a breakpoint at the first exit code statement (which will be `NULL` if `on.exit` has not yet been called); this has the effect of a "run-until-finished-then-pause" breakpoint. Whenever `on.exit` is called, any existing exit code breakpoints are lost; but if any were present, a new unconditional breakpoint is set at the start of the exit code.

Breakpoints are evaluated in step mode too, but the debugger remains in step mode whatever the result.

At present, all breakpoints are destroyed when functions are edited (though if you use `fixr`, `mtrace` will be re-applied automatically). However, the S+ versions of `debug` and `mvbutils` make an effort to preserve breakpoints across edits, and I plan to introduce something similar in R.

### Author(s)

Mark Bravington

### See Also

[mtrace](#), [go](#)

### Examples

```
## Not run:
mtrace( glm)
glm( 35)
# Once the debugger starts:
bp(7) # unconditional breakpoint at line 7
bp(7,F) # to clear it.
```

```

bp(7,x>1) # conditional; will trigger if "x>1" (or if "x>1" causes error)
bp(1,F,"glm") # can be called BEFORE debugging glm;
# prevents debugger from halting at start of function
qqq() # exit debugger
mtrace.off()

## End(Not run)

```

---

fun.locator

*Get environment(s) where an object exists*


---

### Description

Checks the frame stack, the search list, any namespaces, and any S3 method tables for copies of an object (normally a function, but this isn't checked or required). Used by `mtrace`.

### Usage

```
fun.locator( fname, from=.GlobalEnv)
```

### Arguments

<code>fname</code>	character string (the object's name)
<code>from</code>	Where to start looking: either an environment, another function (in which case, that function's defining environment is used), or a frame number. 0 is treated as <code>.GlobalEnv</code>

### Details

When `fname` is defined in a namespaced package **foo**, several copies will exist at once. If `foo` exports `fname`, then `fname` will be found both in the search path (in `package:foo`) and in the "hidden" environment `asNamespace("foo")`. The first version will be found by "normal" calls to `fname`, but the hidden version will be found whenever `fname` is invoked by a function in `package:foo`. If the S3 method `somegeneric.fname` is defined in `package:foo`, then it will exist both in `asNamespace("foo")` and in `get(".__S3MethodsTable__.", asNamespace("foo"))`, although not in the main search path. If `fname` is both exported and an S3 method, then there will be three copies. All such duplicates should probably be identical, and if you want to change any one copy, you probably want to change all of them.

Normally, the search path environment(s) where `fname` is found will be given before (i.e. with lower index than) the hidden namespace environment. However, if `from` is used to start the search in a hidden namespace environment where `fname` exists, then the hidden namespace will be returned first. Duplicated environments are removed from the return list.

I don't know what happens with S4-style methods.

**Value**

A list of environments where `fname` was found, of length 0 if no copies were found. First, any environments between `from` and `topenv( from)` are searched (normally, this means temporary frames wherein `fname` could have been defined; but see **Details**). Next, the search list is checked. Next, all loaded namespaces are checked. Finally, all S3 method tables are checked.

**Author(s)**

Mark Bravington

**See Also**

`getAnywhere`

---

`get.retval`

*Show current return value when debugging*

---

**Description**

When debugging a function, `get.retval` gives the current return value, i.e. the result of the most recent valid statement executed in the function body, whether typed or in the original code, and excluding calls to `go` etc.. Presumably only relevant when in exit code.

**Usage**

```
get.retval()
```

**Details**

To **change** the return value while in exit code, use `skip` to move back into the function code, then `return(whatever)`.

**Author(s)**

Mark Bravington

**See Also**

`skip`, `go`

---

`go-skip-qqq`*Flow control for debugger*

---

### Description

`go`, `skip` and `qqq` ONLY work inside the debugger, i.e. while you are paused at a `D(..)>` prompt during the execution of a function that has been `mtrace`d. `go` makes the debugger begin executing code continuously, without user intervention; `skip(n)` shifts the execution point; `qqq()` quits the debugger.

### Usage

```
go(line.no) # line.no can be missing
skip(line.no)
qqq()
```

### Arguments

`line.no`      a line number, as shown in the code window (see also DETAILS)

### Details

`go()` without any argument puts the debugger into "go mode", whereby it starts executing function code without pausing for input (see `package?debug`). `go(n)` basically means "run continuously until you reach line `n`". It sets a temporary breakpoint at line `n`, which is triggered the first time line `n` is reached and then immediately cleared.

`skip(n)` moves the execution point (highlighted in the code window) to line `n`, without executing any intervening statements. You can `skip` forwards and backwards, and between the main function code and the exit code. You can `skip` in and out of loops and conditionals, except that you can't skip into a `for` loop (the execution point will move to the start of the loop instead). Note that skipping backwards does not undo any statements already executed. `skip` is useful for circumventing errors, and for ensuring that exit code gets run before calling `qqq()`.

`qqq()` quits the debugger, closing all code windows, and returns to the command prompt. No further code statements will be executed, which means no exit code either; take care with open files and connections.

### Author(s)

Mark Bravington

### See Also

package `debug`, `mtrace`, `bp`

---

`is.mtraced`
*Check if a function has been 'mtrace'd.*


---

**Description**

Check if a function has been `mtraced`.

**Usage**

```
is.mtraced(f)
```

**Arguments**

`f` a function

**Value**

TRUE or FALSE.

**See Also**

`mtrace`

**Examples**

```
fff <- function() 99
is.mtraced( fff)
mtrace( fff)
is.mtraced( fff)
```

---

`mtrace`
*Interactive debugging*


---

**Description**

`mtrace` sets or clears debugging mode for a function; `mtrace.off` clears debugging mode for all functions; `check.for.tracees` shows which functions are in debugging mode.

**Usage**

```
mtrace( fname, tracing=TRUE, char.fname, from=mvb.sys.parent(), update.tracees=TRUE)
# Usual: mtrace( fname) or mtrace( fname, tracing=FALSE, return.envs=FALSE)
mtrace.off()
check.for.tracees( where=1)
```

## Arguments

<code>fname</code>	quoted or unquoted function name
<code>tracing</code>	TRUE to turn tracing on, FALSE to turn it off
<code>char.fname</code>	(rarely used) if your function name is stored in a character object <code>x</code> , use <code>char.fname=x</code>
<code>from</code>	where to start looking for <code>fname</code> (not usually needed)
<code>where</code>	(character or integer) position in search path
<code>update.tracees</code>	don't set this parameter! It's only for use by other functions
<code>return.envs</code>	if TRUE, this will return a list of the environments where the function has been replaced by the <code>mtraced</code> version

## Details

`mtrace(myfun)` modifies the body code of `myfun`, and also stores debugging information about `myfun` in `tracees$myfun`. Next time the function is invoked, the modified debug-ready version will be called instead of the original. `mtrace` does not modify source code (or other) attributes, so `myfun` will "look" exactly the same afterwards. `mtrace(myfun, FALSE)` restores `myfun` to normal. `mtrace.off` unmtraces all mtraced functions (see below for exceptions).

Because `mtrace` modifies function bodies (possibly in several places, if namespaced packages are involved), calling `save.image` or `save` while functions are still mtraced is probably not a good idea— if the saved image is reloaded in a new R session, the debugger won't know how to handle the previously mtraced functions, and an error message will be given if they are invoked. The `Save` and `Save.pos` functions in package **mvbutils** will get round this without your having to manually untrace and retrace functions.

If you do see a "maybe saved before being un-mtraced?" error message when `myfun` is invoked, all is not lost; you can restore `myfun` to health via `mtrace(myfun, F)`, or put it properly into debugging mode via `mtrace(myfun).mtrace.off` won't work in such cases, because `myfun` isn't included in `tracees`.

`mtrace` looks for a function in the following places: first in the frame stack (unless `from` has been set— see below), then in the search path, then in all namespaces, then in all S3 methods tables. If several copies of a function are found, all will get modified (mtraced) to the **same** code; ditto when unmtracing.

`mtrace` etc **doesn't** (currently) handle S4 methods. However, I doubt there's any fundamental problem there (hopefully I will never have to find out)— the first issue, though, is that S4 methods have mangled names. If you can work out what the mangled name is, then you can mtrace the "real" version and all may work.

`check.for.tracees` checks for functions which have been mtraced, but only in one directory. By contrast, `names(tracees)` will return all functions that are currently known to be mtraced. However, unlike `check.for.tracees`, `names(tracees)` won't show functions that were saved during a previous R session in an mtraced state.

`mtrace.off` will untrace all functions returned by `names(tracees)` and/or `check.for.tracees(1)`.

`mtrace` puts a breakpoint (see [bp](#)) at line 1, but clears all other breakpoints.

mtrace can handle mlocal functions, but not (yet) do.in.envir functions— the latter appear as monolithic statements in the code window. See package **mvbutils** for more details.

If you use `fixr` to edit functions, mtrace will automatically be re-applied when an updated function file is sourced back in. Otherwise, you'll need to call `mtrace` manually after updating a function.

The `from` argument is only used in the following situations. The first is to mtrace an S4 method that lives somewhere. Suppose there is a function `f` which first defines functions `g` and `h`, then calls `g`. Now suppose you have mtraced `g` and that `g` is running, with the code window displayed. If you want to mtrace(`h`), the problem is that `h` is not visible from the frame of `g`. To tell mtrace where to find `g`, call `mtrace(h, from=sys.parent())`. [You can also replace `sys.parent()` with the absolute frame number of `f`, if `f` has been mtraced and its code window is visible.] mtrace will then look through the enclosing environments of `from` until it finds a definition of `h`.

If `myfun` has been defined in a namespaced package, then there may be several copies of `myfun` in the system, different ones being used at different times. mtrace will change them all; see `fun.locator` if you really want to know more.

If `mtrace(bar)` is called while function `foo` is being debugged (`mtrace(foo)` having previously been called), and `bar` has been redefined within `foo` or a parent environment of `foo`, then only the redefined copy of `bar` will be mtraced.

## Value

Not normally used. mtrace by default returns an invisible copy of the modified function body. If you set `return.envs=TRUE`, it will instead return a list of the environments in which the function has been modified. This is only intended for "internal use".

## Examples

```
## Not run:
mtrace(glm) # turns tracing on
names( tracees) # "glm"
check.for.tracees( "package:base") # "glm"
glm(stupid.args) # voila le debugger
qqq() # back to command prompt
mtrace( glm, FALSE)
mtrace.off() # turns it off for all functions

## End(Not run)
```

# Index

## \*Topic **debugging**

- bp, [5](#)
- debug-package, [2](#)
- get.retval, [8](#)
- go-skip-qqq, [9](#)
- is.mtraced, [10](#)
- mtrace, [10](#)

## \*Topic **programming**

- fun.locator, [7](#)

## \*Topic **utilities**

- fun.locator, [7](#)

bp, [3](#), [5](#), [5](#), [9](#), [11](#)

check.for.tracees, [5](#)  
check.for.tracees (*mtrace*), [10](#)

debug (*debug-package*), [2](#)  
debug-package, [2](#)

fun.locator, [7](#), [12](#)

get.retval, [5](#), [8](#)  
go, [5](#), [6](#), [8](#)  
go (*go-skip-qqq*), [9](#)  
go-skip-qqq, [9](#)

is.mtraced, [10](#)

mtrace, [3](#), [5-7](#), [9](#), [10](#), [10](#)  
mtrace.off, [5](#)

qqq, [5](#)  
qqq (*go-skip-qqq*), [9](#)

skip, [3](#), [5](#), [8](#)  
skip (*go-skip-qqq*), [9](#)

tracees (*mtrace*), [10](#)