Debugging with `ruby-debug`

0.10.4 Edition
October 2010

Rocky Bernstein and Kent Sibilev

Table of Contents

1 Summary of `ruby-debug`

The purpose of a debugger such as ruby-debug is to allow you to see what is going on “inside” a Ruby program while it executes.

`rdebug` can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your script, specifying anything that might affect its behavior.
- Make your script stop on specified conditions.
- Examine what has happened, when your script has stopped.
- Change things in your script, so you can experiment with correcting the effects of one bug and go on to learn about another.

Although you can use `rdebug` to invoke your Ruby programs via a debugger at the outset, there are other ways to use and enter the debugger.

1.1 The First Sample `rdebug` Session (`list`, `display`, `print`, and `quit`)

You can use this manual at your leisure to read all about `ruby-debug`. However, a handful of commands are enough to get started using the debugger. The following sections illustrates these commands.

In this sample session, we emphasize user input like this: **input**, to make it easier to pick out from the surrounding output.

Below is Ruby code to compute a triangle number of a given length.[1]

```
$ rdebug triangle.rb
triangle.rb:4 def hanoi(n,a,b,c)
(rdb:1) list
[-1, 8] in ./triangle.rb
   1  #!/usr/bin/env ruby
   2  # Compute the n'th triangle number - the hard way
   3  # triangle(n) == (n * (n+1)) / 2
=> 4  def triangle(n)
   5    tri = 0
   6    0.upto(n) do |i|
   7      tri += i
   8    end
(rdb:1) l
[9, 18] in ./triangle.rb
   9    return tri
   10  end
   11
   12  puts triangle(3)
(rdb:1) list 1,100
[1, 100] in ./triangle.rb
   1  #!/usr/bin/env ruby
   2  # Compute the n'th triangle number - the hard way
```

[1] There are of course shorter ways to define `triangle` such as:

```
def triangle(n) (n * (n+1)) / 2 end
```

The code we use in this example and the next is more for pedagogical purposes than how to write short Ruby code.

```
   3  # triangle(n) == (n * (n+1)) / 2
=> 4  def triangle(n)
   5    tri = 0
   6    0.upto(n) do |i|
   7      tri += i
   8    end
   9    return tri
   10  end
   11
   12  puts triangle(3)
(rdb:1)
```

There are lots of command options, but we don't need them for now. See Section 2.1.1 [rdebug command-line options], page 17 for a full list of command options.

Position information consists of a filename and line number, e.g. `triangle.rb:4`. We are currently stopped before the first executable line of the program; this is line 4 of `triangle.rb`. If you are used to less dynamic languages and have used debuggers for more statically compiled languages like C, C++, or Java, it may seem odd to be stopped before a function definition. But in Ruby line 4 is executed, the name `triangle` (probably) does not exist so issuing a method call of `triangle` will raise a "method not found" error.

ruby-debug's prompt is `(rdb:`*n*`)`. The *n* is the thread number. Here it is 1 which is usually the case for the main thread. If the program has died and you are in post-mortem debugging, there is no thread number. In this situation, the string `post-mortem` is used in place of a thread number. If the program has terminated normally, the string this position will be `ctrl`. The commands which are available change depending on the program state.

The first command, `list` (see Section 3.9 [List], page 30), prints 10 lines centered around the current line; the current line here is line 4 and is marked by `=>`, so the range the debugger would like to show is -1..8. However since there aren't 5 lines before the current line, those additional lines—"lines" -1 and 0—are dropped and we print the remaining 8 lines. The `list` command can be abbreviated with `l` which is what we use next. Notice that when we use this a second time, we continue listing from the place we last left off. The desired range of lines this time is lines 9 to 18; but since the program ends as line 12, only the remaining 4 lines are shown.

If you want to set how many lines to print by default rather than use the initial number of lines, 10, use the `set listsize` command (see Section 3.13.14 [Listsize], page 40). To see the entire program in one shot, we gave an explicit starting and ending line number.

If you use a front-end to the debugger such as the Emacs interface, you probably won't use `list` all that much.

Now let us step through the program.

```
(rdb:1) step
triangle.rb:12
puts triangle(3)
(rdb:1) <RET>
triangle.rb:5
tri = 0
(rdb:1) p tri
nil
(rdb:1) step
triangle.rb:6
0.upto(n) do |i|
(rdb:1) p tri
0
```

The first *step* command (see Section 3.12.4.1 [Step], page 36) runs the script one executable unit. The second command we entered was just hitting the return key; `rdebug` remembers the last command you entered was `step`, so it runs that last command again.

One way to print the values of variables uses `p`. (Of course, there are of course lots of other ways too.). When we look at the value of `tri` the first time, we see it is `nil`. Again we are stopped *before* the assignment on line 5, and this variable hasn't been set previously. However after issuing another "step" command we see that the value is 0 as expected. You could issue the step and print comman in one shot:

However if every time we stop we want to see the value of `tri` to see how things are going stop, there is a better way by setting a display expression (see Section 3.6 [Display-Commands], page 26).

```
(rdb:1) display tri
1: tri = 0
```

Now let us run the program until we return from the function. However we'll want to see which lines get run.

```
(rdb:1) display i
2: i =
(rdb:1) set linetrace on
line tracing is on.
(rdb:1) finish
Tracing(1):triangle.rb:7 tri += i
1: tri = 0
2: i = 0
Tracing(1):triangle.rb:7 tri += i
1: tri = 0
2: i = 1
Tracing(1):triangle.rb:7 tri += i
1: tri = 1
2: i = 2
Tracing(1):triangle.rb:7 tri += i
1: tri = 3
2: i = 3
Tracing(1):triangle.rb:9 return tri
1: tri = 6
2: i =
(rdb:1) quit
Really quit? (y/n) y
```

So far, so good. A you can see from the above to get out of the debugger, one can issue a `quit` command. (`q` and `exit` are just as good. If you want to quit without being prompted, suffix the command with an exclamation mark, e.g.`\q!`.

1.2 Sample Session 2: Delving Deeper (`where`, `frame`, `restart`, `autoeval`, `break`, `ps`)

In this section we'll introduce breakpoints, the call stack and restarting. So far we've been doing pretty good in that we've not encountered a bug to fix. Let's try another simple example. Okay here's the program.

Below we will debug a simple Ruby program to solve the classic Towers of Hanoi puzzle. It is augmented by the bane of programming: some command-parameter processing with error checking.

```
$ rdebug hanoi.rb
hanoi.rb:3 def hanoi(n,a,b,c)
(rdb:1) list 1,100
[1, 100] in ./hanoi.rb
   1  #!/usr/bin/ruby
   2
=> 3  def hanoi(n,a,b,c)
   4      if n-1 > 0
   5         hanoi(n-1, a, c, b)
   6      end
   7      puts "Move disk %s to %s" % [a, b]
   8      if n-1 > 0
   9         hanoi(n-1, c, b, a)
   10      end
   11  end
   12
   13  i_args=ARGV.length
   14  if i_args > 1
   15      puts "*** Need number of disks or no parameter"
   16      exit 1
   17  end
   18
   19  n=3
   20
   21  if i_args > 0
   22      begin
   23        n = ARGV[0].to_i
   24      rescue ValueError, msg:
   25          print "** Expecting an integer, got: %s" % ARGV[0].to_s
   26        exit 2
   27      end
   28  end
   29
   30  if n < 1 or n > 100
   31      puts "*** number of disks should be between 1 and 100"
   32      exit 2
   33  end
   34
   35  hanoi(n, :a, :b, :c)
(rdb:1)
```

Recall in the first section I said that before the `def` is run the method it names is undefined. Let's check that out. First let's see what private methods we can call before running `def hanoi`

```
(rdb:1) set autoeval on
autoeval is on.
(rdb:1) private_methods
["select", "URI", "local_variables", "lambda", "chomp", ...
```

The `set autoeval` (see Section 3.13.2 [Autoeval], page 38) command causes any commands that are not normally understood to be debugger commands to get evaluated as though they were Ruby commands. I use this a lot, so I set this by putting it the command file `.rdebugrc`, see Section 2.2 [Command Files], page 20, that gets read when `ruby-debug` starts.

As showing the list output of `private_methods`, I find this kind of list unwieldy. What you are supposed to notice here is that method `hanoi` is not in this list. When you ask `ruby-debug` for a list of method names via `method instance`, it doesn't show output in this way; `ruby-debug` can sort and put into columns lists like this using the print command, `ps`.

```
(rdb:1) ps private_methods
Array                    exit!                puts                        warn
Float                    fail                 raise                       y
Integer                  fork                 rand
Rational                 format               readline
String                   gem_original_require readlines
URI                      getc                 remove_instance_variable
`                        gets                 scan
abort                    global_variables     select
active_gem_with_options  gsub                 set_trace_func
at_exit                  gsub!                singleton_method_added
autoload                 initialize           singleton_method_removed
autoload?                initialize_copy      singleton_method_undefined
binding                  iterator?            sleep
block_given?             lambda               split
callcc                   load                 sprintf
caller                   local_variables      srand
catch                    location_of_caller   sub
chomp                    loop                 sub!
chomp!                   method_missing       syscall
chop                     open                 system
chop!                    p                    test
dbg_print                pp                   throw
dbg_puts                 print                timeout
eval                     printf               trace_var
exec                     proc                 trap
exit                     putc                 untrace_var
```

Now let's see what happens after stepping

```
(rdb:1) private.methods.member?("hanoi")
false
(rdb:1) step
hanoi.rb:13
i_args=ARGV.length
(rdb:1) private_methods.member?("hanoi")
true
(rdb:1)
```

Okay, now where were we?

```
(rdb:1) list
[8, 17] in ./hanoi.rb
   8      if n-1 > 0
   9         hanoi(n-1, c, b, a)
   10      end
   11  end
   12
=> 13  i_args=ARGV.length
   14  if i_args > 1
   15      puts "*** Need number of disks or no parameter"
   16      exit 1
   17  end
(rdb:1) ARGV
[]
```

Ooops. We forgot to specify any parameters to this program. Let's try again. We can use the `restart` command here.

```
(rdb:1) restart 3
Re exec'ing:
        /usr/bin/rdebug hanoi.rb 3
hanoi.rb:3
def hanoi(n,a,b,c)
(rdb:1) break 4
Breakpoint 1 file hanoi.rb, line 4
(rdb:1) continue
Breakpoint 1 at hanoi.rb:4
./hanoi.rb:4 if n-1 > 0
(rdb:1) display n
1: n = 3
(rdb:1) display a
2: a = a
(rdb:1) undisplay 2
(rdb:1) display a.inspect
3: a.inspect = :a
(rdb:1) display b.inspect
4: b.inspect = :b
(rdb:1) continue
Breakpoint 1 at hanoi.rb:4
./hanoi.rb:4
if n-1 > 0
1: n = 2
3: a.inspect = :a
4: b.inspect = :c
(rdb:1) c
Breakpoint 1 at hanoi.rb:4
./hanoi.rb:4
if n-1 > 0
1: n = 1
3: a.inspect = :a
4: b.inspect = :b
(rdb:1) where
--> #0 Object.hanoi(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at line hanoi.rb:4
    #1 Object.-(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at line hanoi.rb:5
    #2 Object.-(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at line hanoi.rb:5
    #3 at line hanoi.rb:35
(rdb:1)
```

In the above we added a new command, `break` (see Section 3.12.1 [Breakpoints], page 33) which indicates to go into the debugger just before that line of code is run. And `continue` resumes execution. Notice the difference between `display a` and `display a.inspect`. An implied string conversion is performed on the expression after it is evaluated. To remove a display expression we used `undisplay` is used. If we give a display number, just that display expression is removed.

Above we also used a new command `where` (see Section 3.11.2 [Backtrace], page 32 to show the call stack. In the above situation, starting from the bottom line we see we called the hanoi from line 35 of the file `hanoi.rb` and the hanoi method called itself two more times at line 5.

In the call stack we show the file line position in the same format when we stop at a line. Also we see the names of the parameters and the types that those parameters *currently* have. It's possible that when the program was called the parameter had a different type, since the types of variables can change dynamically. You alter the style of what to show in the trace (see Section 3.13.7 [Callstyle], page 39).

Let's explore a little more. Now were were we?

```
(rdb:1) list
   1  #!/usr/bin/ruby
   2
   3  def hanoi(n,a,b,c)
=> 4      if n-1 > 0
   5         hanoi(n-1, a, c, b)
   6      end
   7      puts "Move disk %s to %s" % [a, b]
   8      if n-1 > 0
(rdb:1) undisplay
Clear all expressions? (y/n) y
(rdb:1) i_args
NameError Exception: undefined local variable or method `i_args' for main:Object
(rdb:1) frame -1
#3 at line hanoi.rb:35
(rdb:1) i_args
1
(rdb:1) p n
3
(rdb:1) down 2
#2 Object.-(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at line hanoi.rb:5
(rdb:1) p n
2
```

Notice in the above to get the value of variable `n`, I have to use a print command like `p n`; If I entered just `n`, that would be taken to mean the debugger command "next". In the current scope, variable `i_args` is not defined. However I can change to the top-most frame by using the `frame` command. Just as with arrays, -1 means the last one. Alternatively using frame number 3 would have been the same thing; so would issuing `up 3`.

Note that in the outside frame 3, the value of `i_args` can be shown. Also note that the value of variable `n` is different.

1.3 Using the debugger in unit testing (ruby-debug/debugger, Debugger.start)

In the previous sessions we've been calling the debugger right at the outset. I confess that this mode of operation is usually not how I use the debugger.

There are a number of situations where calling the debugger at the outset is impractical for a couple of reasons.

1. The debugger just doesn't work when run at the outset. By necessity any debugging changes to the behavior or the program in slight and subtle ways, and sometimes this can hinder finding the bugs.
2. There's a lot of code which that needs to get run before the part you want to inspect. Running this code takes time and you don't the overhead of the debugger in this first part.

In this section we'll delve show how to enter the code in the middle of your program, while delving more into the debugger operation.

In this section we will also use unit testing. Using unit tests will greatly reduce the amount of debugging needed while at the same time increase the quality of your program.

What we'll do is take the `triangle` code from the first session and write a unit test for that. In a sense we did write a mini-test for the program which was basically the last line where we printed the value of triangle(3). This test however wasn't automated: the implication is that someone would look at the output and verify that what was printed is what was expected.

And before we can turn that into something that can be `required`, we probably want to remove that output. However I like to keep in that line so that when I look at the file, I have an example of how to run it. Therefore we will conditionally run this line if that file is invoked directly, but skip it if it is not.[2]

```
if __FILE__ == $0
  puts triangle(3)
end
```

Let's call this file `tri2.rb`.

Okay, we're now ready to write our unit test. We'll use `"test/unit"` which comes with the standard Ruby distribution. Here's the test code:

```
#!/usr/bin/env ruby
require 'test/unit'
require 'tri2.rb'

class TestTri < Test::Unit::TestCase
  def test_basic
   solutions = []
   0.upto(5) do |i|
      solutions << triangle(i)
    end
    assert_equal([0, 1, 3, 6, 10, 15], solutions,
                 'Testing the first 5 triangle numbers')
  end
end
```

If you run it will work. However if you run `rdebug` initially, you will not get into the test, because `test/unit` wants to be the main program. So here is a situation where one may need to modify the program to add an explicit *entry* into the debugger.[3]

One way to do this is to add the following before the place you want to stop:

```
require 'rubygems'
require 'ruby-debug/debugger'
```

The line `require "rubygems"` is needed if `ruby-debug` is installed as a Ruby gem.

Let's add this code just after entering `test_basic`:

```
...
def test_basic
  require "rubygems"
  require "ruby-debug/debugger"
  solutions = []
 ...
```

Now we run the program..

[2] `rdebug` resets `$0` to try to make things like this work.

[3] For some versions of rake and `rdebug` you can in fact set a breakpoint after running `rdebug` initially. Personally though I find it much simpler and more reliable to modify the code as shown here.

```
    $ ruby test-tri.rb
    Loaded suite test-tri
    Started
    test-tri.rb:9
    solutions = []
    (rdb:1)
```

and we see that we are stopped at line 9 just before the initialization of the list `solutions`.

Now let's see where we are...

```
(rdb:1) where
--> #0 TestTri.test_basic at line /home/rocky/ruby/test-tri.rb:9
(rdb:1)
```

Something seems wrong here; `TestTri.test_basic` indicates that we are in class `TestTri` in method `test_basic`. However we don't see the call to this like we did in the last example when we used the `where` command. This is because the debugger really didn't spring into existence until after we already entered that method, and Ruby doesn't keep call stack information around in a way that will give the information we show when running `where`.

If we want call stack information, we have to turn call-stack tracking on *beforehand*. This is done by adding `Debugger.start`.

Here's what our test program looks like so after we modify it to start tracking calls from the outset

```
#!/usr/bin/env ruby
require 'test/unit'
require 'tri2.rb'
require 'rubygems'
Debugger.start

class TestTri < Test::Unit::TestCase
  def test_basic
    debugger
    solutions = []
    0.upto(5) do |i|
      solutions << triangle(i)
    end
    assert_equal([0, 1, 3, 6, 10, 15], solutions,
                 "Testing the first 5 triangle numbers")
  end
end
```

Now when we run this:

```
$ ruby test-tri2.rb
Loaded suite test-tri2
Started
test-tri2.rb:11
solutions = []
(rdb:1) where
--> #0 TestTri.test_basic at line test-tri2.rb:11
    #1 Kernel.__send__(result#Test::Unit::TestResult)
       at line /usr/lib/ruby/1.8/test/unit/testcase.rb:70
    #2 Test::Unit::TestCase.run(result#Test::Unit::TestResult)
       at line /usr/lib/ruby/1.8/test/unit/testcase.rb:70
...
    #11 Test::Unit::AutoRunner.run
       at line /usr/lib/ruby/1.8/test/unit/autorunner.rb:200
    #12 Test::Unit::AutoRunner.run(force_standalone#FalseClass, ...
       at line /usr/lib/ruby/1.8/test/unit/autorunner.rb:13
    #13 at line /usr/lib/ruby/1.8/test/unit.rb:285
(rdb:1)
```

Much better. But again let me emphasize that the parameter types are those of the corresponding variables that *currently* exist, and this might have changed since the time when the call was made. Even so and even though we only have *types* listed, it's a pretty good bet that when `Test::Unit` was first called, shown above as frame 12, that the values of its two parameters were `false` and `nil`.

1.4 Using the `Debugger.start` with a block

We saw that `Debugger.start()` and `Debugger.stop()` allow fine-grain control over where the debugger tracking should occur.

Rather than use an explicit `stop()`, you can also pass a block to the `start()` method. This causes `start()` to run and then `yield` to that block. When the block is finished, `stop()` is run. In other words, this wraps a `Debugger.start()` and `Debugger.stop()` around the block of code. But it also has a side benefit of ensuring that in the presence of an uncaught exception `stop` is run, without having to explicitly use `begin` ... `ensure Debugger.stop() end`.

For example, in Ruby Rails you might want to debug code in one of the controllers without causing any slowdown to any other code. And this can be done by wrapping the controller in a `start()` with a block; when the method wrapped this way finishes the debugger is turned off, and the application proceeds at regular speed.

Of course, inside the block you will probably want to enter the debugger using `Debugger.debugger()`, otherwise there would little point in using the `start`. For example, you can do this in `irb`:

```
$ irb
irb(main):001:0> require flrubygemsfl; require flruby-debugfl
=> true
irb(main):002:0> def foo
irb(main):003:1> x=1
irb(main):004:1> puts flfoofl
irb(main):005:1> end
=> nil
irb(main):006:0> Debugger.start{debugger; foo}
(irb):6
(rdb:1) s
(irb):3
(rdb:1) p x
nil
(rdb:1) s
(irb):4
(rdb:1) p x
1
(rdb:1) s
foo
=> true
irb(main):007:0>
```

There is a counter inside of `Debugger.start` method to make sure that this works when another `Debugger.start` method is called inside of outer one. However if you are stopped inside the debugger, issuing another `debugger` call will not have any effect even if it is nested inside another `Debugger.start`.

1.5 How debugging Ruby may be different than debugging other Languages

If you are used to debugging in other languages like C, C++, Perl, Java or even Bash[4], there may be a number of things that seem or feel a little bit different and may confuse you. A number of these things aren't oddities of the debugger per see, so much as a difference in how Ruby works compared to those other languages. Because Ruby works a little differently from those other languages, writing a debugger has to also be a little different as well if it is to be useful.

In this respect, using the debugger may help you understand Ruby better.

We've already seen two examples of such differences. One difference is the fact that we stop on method definitions or `def`'s and that's because these are in fact executable statements. In other compiled languages this would not happen because that's already been done when you compile the program (or in Perl when it scans in the program). The other difference we saw was in our inability to show call stack parameter types without having made arrangements for the debugger to track this. In other languages call stack information is usually available without asking assistance of the debugger.[5]

In this section we'll consider some other things that might throw off new users to Ruby who are familiar with other languages and debugging in them.

1.5.1 Stack Shows Scope Nesting

In a backtrace, you will find more stack frames than you might in say C.

Consider another way to write the triangle program of see Section 1.1 [First Sample Session], page 1.

```
1 #!/usr/bin/env ruby
2 def triangle(n)
3   (0..n).inject do |sum, i|
4    sum +=i
5   end
6 end
7 puts triangle(3)
```

Let's stop inside the `inject` block:

```
$ rdebug tri3.rb
(rdb:1) c 4
tri3.rb:4
sum +=i
(rdb:1) where
--> #0 Range.triangle at line tri3.rb:4
    #1 Enumerable.inject at line tri3.rb:3
    #2 Object.triangle(n#Fixnum) at line tri3.rb:3
    #3 at line tri3.rb:7
(rdb:1)
```

Because a new scope was entered, it appears as a stack frame. Probably "scope" frame would be a more appropriate name.

---

[4] this is just an excuse to put in a shameless plug for my bash debugger `http://bashdb.sf.net`

[5] However in C, and C++ generally you have to ask the compiler to add such information.

1.5.2 More Frequent Evaluations per Line

Consider this simple program to compute the Greatest Common Divisor of two numbers:

```
 1 #!/usr/bin/env ruby
 2 # GCD. We assume positive numbers
 3
 4 def gcd(a, b)
 5   # Make: a <= b
 6  if a > b
 7     a, b = [b, a]
 8  end
 9
10  return nil if a <= 0
11
12  if a == 1 or b-a == 0
13    return a
14   end
15   return gcd(b-a, a)
16 end
17
18 a, b = ARGV[0..1].map {|arg| arg.to_i}
19 puts "The GCD of %d and %d is %d" % [a, b, gcd(a, b)]
```

Now let's try tracing a portion of the program to see what we get.

```
$ rdebug gcd.rb 3 5
gcd.rb:4
def gcd(a, b)
(rdb:1) step
gcd.rb:18
a, b = ARGV[0..1].map {|arg| arg.to_i}
(rdb:1) step
gcd.rb:18
a, b = ARGV[0..1].map {|arg| arg.to_i}
(rdb:1) step
gcd.rb:18
a, b = ARGV[0..1].map {|arg| arg.to_i}
(rdb:1) step
(rdb:1) break Object.gcd
Breakpoint 1 at Object::gcd
(rdb:1) continue
Breakpoint 1 at Object:gcd
gcd.rb:4
def gcd(a, b)
(rdb:1) set linetrace on
line tracing is on.
(rdb:1) continue
Tracing(1):gcd.rb:6 if a > b
Tracing(1):gcd.rb:6 if a > b
Tracing(1):gcd.rb:10 return nil if a <= 0
Tracing(1):gcd.rb:10 return nil if a <= 0
Tracing(1):gcd.rb:12 if a == 1 or b-a == 0
Tracing(1):gcd.rb:12 if a == 1 or b-a == 0
Tracing(1):gcd.rb:15 return gcd(b-a, a)
Breakpoint 1 at Object:gcd
gcd.rb:4
def gcd(a, b)
(rdb:1)
```

The thing to note here is that we see lots of lines duplicated. For example, the first line:

```
Tracing(1):gcd.rb:18 a, b = ARGV[0..1].map {|arg| arg.to_i}
```

appears three times. If we were to break this line into the equivalent multi-line expression:

```
a, b = ARGV[0..1].map do |arg|
  arg.to_i
end
```

we would find one stop at the first line before running `map` and two listings of `arg.to_i`, once for each value of arg which here is 0 and then 1. Perhaps this is is not surprising because we have a loop here which gets run in this situation 3 times. A similar command `next`, can also be used to skip over loops and method calls.

But what about all the duplicated `if` statements in `gcd`? Each one is listed twice whether or not we put the `if` at the beginning or the end. You will find this to be the case for any conditional statement such as `until` or `while`.

Each statement appears twice because we stop once before the expression is evaluated and once after the expression is evaluated but before the if statement takes hold. There is a bug in Ruby up to version 1.8.6 in that we stop a second time before the evaluation, so examining values that may have changed during the expression evaluation doesn't work in these versions.

If you are issuing a `step` command one at a time, the repetitive nature can be little cumbersome if not annoying. So ruby-debug offers a variant called `step+` which forces a new line on every step. Let's try that.

```
(rdb:1) R
Re exec'ing:
/usr/bin/rdebug gcd.rb 3 5
gcd.rb:4
def gcd(a, b)
(rdb:1) step+
gcd.rb:18
a, b = ARGV[0..1].map {|arg| arg.to_i}
(rdb:1) step+
gcd.rb:19
puts "The GCD of %d and %d is %d" % [a, b, gcd(a, b)]
(rdb:1) break Object.gcd
Breakpoint 1 at Object:gcd
(rdb:1) c
Breakpoint 1 at Object:gcd
gcd.rb:4
def gcd(a, b)
(rdb:1) set linetrace+
line tracing style is different consecutive lines.
(rdb:1) set linetrace on
line tracing is on.
(rdb:1) c
Tracing(1):gcd.rb:6 if a > b
Tracing(1):gcd.rb:10 return nil if a <= 0
Tracing(1):gcd.rb:12 if a == 1 or b-a == 0
Tracing(1):gcd.rb:15 return gcd(b-a, a)
Breakpoint 1 at Object:gcd
gcd.rb:4
def gcd(a, b)
```

If you want `step+` to be the default behavior when stepping, issue the command `set forcestep on`, (see Section 3.13.8 [Forcestep], page 39). I generally put this in my start-up file `.rdebugrc`.

Similar to the difference between `step+` and `step` is `set linetrace+`. This removes duplicate consecutive line tracing.

One last thing to note above is the use of a method name to set a breakpoint position, rather than a file and line number. Because method `gcd` is in the outermost scope, we use `Object` as the class name.

1.5.3 Bouncing Around in Blocks (e.g. Iterators)

When debugging languages with coroutines like Python and Ruby, a method call may not necessarily go to the first statement after the method header. It's possible the call will continue after a `yield` statement from a prior call.

```
 1 #!/usr/bin/env ruby
 2 # Enumerator for primes
 3 class SievePrime
 4   @@odd_primes = []
 5   def self.next_prime(&block)
 6     candidate = 2
 7     yield candidate
 8     not_prime = false
 9     candidate += 1
10     while true do
11       @@odd_primes.each do |p|
12         not_prime = (0 == (candidate % p))
13         break if not_prime
14       end
15       unless not_prime
16         @@odd_primes << candidate
17         yield candidate
18       end
19       candidate += 2
20     end
21   end
22 end
23 SievePrime.next_prime do |prime|
24   puts prime
25   break if prime > 10
26 end
```

```
$ rdebug primes.rb
primes.rb:3
class SievePrime
(rdb:1) set linetrace on
line tracing is on.
(rdb:1) step 10
Tracing(1):primes.rb:4 @odd_primes = []
Tracing(1):primes.rb:5 def self.next_prime(&block)
Tracing(1):primes.rb:23 SievePrime.next_prime do |prime|
Tracing(1):primes.rb:6 candidate = 2
Tracing(1):primes.rb:7 yield candidate
Tracing(1):primes.rb:24 puts prime
2
Tracing(1):primes.rb:25 break if prime > 10
Tracing(1):primes.rb:25 break if prime > 10
Tracing(1):primes.rb:8 not_prime = false
Tracing(1):primes.rb:9 candidate += 1
primes.rb:9
candidate += 1
```

```
(rdb:1)
```

The loop between lines 23–26 gets interleaved between those of `Sieve::next_prime`, lines 6–19 above.

A similar kind of thing can occur in debugging programs with many threads.

1.5.4 No Parameter Values in a Call Stack

In traditional debuggers in a call stack you can generally see the names of the parameters and the values that were passed in.

Ruby is a very dynamic language and it tries to be efficient within the confines of the language definition. Values generally aren't taken out of a variable or expression and pushed onto a stack. Instead a new scope created and the parameters are given initial values. Parameter passing is by *reference*, not by value as it is say Algol, C, or Perl. During the execution of a method, parameter values can change—and often do. In fact even the *class* of the object can change.

So at present, the name of the parameter shown. The call-style setting see Section 3.13.7 [Callstyle], page 39 can be used to set whether the name is shown or the name and the *current* class of the object.

It has been contemplated that a style might be added which saves on call shorter "scalar" types of values and the class name.

1.5.5 Lines You Can Stop At

As with the duplicate stops per control (e.g. `if` statement), until tools like debuggers get more traction among core ruby developers there are going to be weirdness. Here we describe the stopping locations which effects the breakpoint line numbers you can stop at.

Consider the following little Ruby program.

```
'Yes it does' =~ /
(Yes) \s+
it  \s+
does
/ix
puts $1
```

The stopping points that Ruby records are the last two lines, lines 5 and 6. If you run `ruby -rtracer` on this file you'll see that this is so:

```
$ ruby -rtracer lines.rb
#0:lines.rb:5::-: /ix
#0:lines.rb:6::-: puts $1
#0:lines.rb:6:Kernel:>: puts $1
#0:lines.rb:6:IO:>: puts $1
Yes#0:lines.rb:6:IO:<: puts $1
#0:lines.rb:6:IO:>: puts $1

#0:lines.rb:6:IO:<: puts $1
#0:lines.rb:6:Kernel:<: puts $1
```

Inside `ruby-debug` you an get a list of stoppable lines for a file using the `info file` command with the attribute `breakpoints`.

2 Getting in and out

It is also possible to enter the debugger when you have an uncaught exception. See See also Chapter 4 [Post-Mortem Debugging], page 43.

2.1 Starting the debugger

Although one can enter ruby-debug via Emacs (described in a later section) and possibly others interfaces, probably the most familiar thing to do is invoke the debugger from a command line.

A wrapper shell script called `rdebug` basically `require`'s the gem package `ruby-debug` and then loads `rdebug`.

```
rdebug [rdebug-options] [--] ruby-script ruby-script-arguments...
```

If you don't need to pass dash options to your program which might get confused with the debugger options, then you don't need to add the '`--`'.

To get a brief list of options and descriptions, use the `--help` option.

```
$ rdebug –help
rdebug 0.10.4
Usage: rdebug [options] <script.rb> -- <script.rb parameters>

Options:
    -A, --annotate LEVEL             Set annotation level
    -c, --client                     Connect to remote debugger
        --cport PORT                 Port used for control commands
    -d, --debug                      Set $DEBUG=true
        --emacs                      Activates full Emacs support
        --emacs-basic                Activates basic Emacs mode
    -h, --host HOST                  Host name used for remote debugging
    -I, --include PATH               Add PATH to $LOAD_PATH
        --keep-frame-binding         Keep frame bindings
    -m, --post-mortem                Activate post-mortem mode
        --no-control                 Do not automatically start control thread
        --no-quit                    Do not quit when script finishes
        --no-rewrite-program         Do not set $0 to the program being debugged
        --no-stop                    Do not stop when script is loaded
    -p, --port PORT                  Port used for remote debugging
    -r, --require SCRIPT             Require the library, before executing your script
        --script FILE                Name of the script file to run
    -s, --server                     Listen for remote connections
    -w, --wait                       Wait for a client connection, implies -s option
    -x, --trace                      Turn on line tracing

Common options:
        --verbose                    Turn on verbose mode
        --help                       Show this message
        --version                    Print the version
    -v                               Print version number, then turn on verbose mode
```

Options for the `rdebug` are shown in the following list.

2.1.1 Options you can pass to rdebug

You can run ruby-debug in various alternative modes—for example, as a program that interacts directly with the program in the same process on the same computer or via a socket to another process possibly on a different computer.

Many options appear as a long option name, such as '`--help`', and a short one letter option name, such as '`-h`'. A double dash ('`--`' is used to separate options which go to `rdebug` from options that are intended to go to your Ruby script. Options (if any) to `rdebug` should come first. If there is no possibility of the Ruby script to be debugged getting confused with `rdebug`'s option the double dash can be omitted.

`--help` This option causes `rdebug` to print some basic help and exit.

`-v | --version`
: This option causes `rdebug` to print its version number and exit.

`-A | --annotate` *`level`*
: Set gdb-style annotation *level*, a number. Additional information is output automatically when program state is changed. This can be used by front-ends such as GNU Emacs to post this updated information without having to poll for it.

`-c | --client`
: Connect to remote debugger. The remote debugger should have been set up previously our you will get a connection error and `rdebug` will terminate.

`--cport` *`port`*
: Port used for control commands.

`--debug` Set `$DEBUG` to `true`. This option is compatible with Ruby's.

`--emacs` Activates GNU Emacs mode. Debugger output is tagged in such a way to allow GNU Emacs to track where you are in the code.

`--emacs-basic`
: Activates full GNU Emacs mode. This is the equivalent of setting the options '`--emacs-basic`', `annotate=3`, '`--no-stop`', '`-no-control`' and '`--post-mortem`'.

`-h | --host` *`host-address`*
: Connect host address for remote debugging.

`-I --include` *`PATH`*
: Add *PATH* to `$LOAD_PATH`

`--keep-frame-binding`
: Bindings are used to set the proper environment in evaluating expression inside the debugger. Under normal circumstances, I don't believe most people will ever need this option.

  By default, the debugger doesn't create binding object for each frame when the frame is created, i.e. when a call is performed. Creating a binding is an expensive operation and has been a major source of performance problems.

  Instead, the debugger creates a binding when there is a need to evaluate expressions. The artificial binding that is created might be different from the real one. In particular, in performing constant and module name resolution.

  However it's still possible to restore the old, slower behavior by using this option or by setting `Debugger.keep_frame_binding = true`. There are two possibilities for which you might want to use this option.

First, if you think there's a bug in the evaluation of variables, you might want to set this to see if this corrects things.

Second, since the internal structures that are used here `FRAME` and `SCOPE` are not part of the Ruby specification, it is possible they can change with newer releases; so here this option this may offer a remedy. (But you'll probably also have to hack the C code since it's likely under this scenario that ruby-debug will no longer compile.) In fact, in Ruby 1.9 these structures have changed and that is partly why this debugger doesn't work on Ruby 1.9.

`-m | --post-mortem`
: If your program raises an exception that isn't caught you can enter the debugger for inspection of what went wrong. You may also want to use this option in conjunction with '`--no-stop`'. See also Chapter 4 [Post-Mortem Debugging], page 43.

`--no-control`
: Do not automatically start control thread.

`--no-quit`
: Restart the debugger when your program terminates normally.

`--no-rewrite-program`
: Normally `rdebug` will reset the program name `$0` from its name to the debugged program, and set the its name in variable `$RDEBUG_0`. In the unlikely even you don't want this use this option.

`--no-stop`
: Normally the `rdebug` stops before executing the first statement. If instead you want it to start running initially and will perhaps break it later in the running, use this options.

`-p | --port` *`port`*
: Port used for remote debugging.

`-r | --require` *`library`*
: Require the library, before executing your script. However if the library happened to be `debug`, we'll just ignore the require (since we're already a debugger). This option is compatible with Ruby's.

`--script` *`file`*
: Require the library, before executing your script. However if the library happend to be `debug`, we'll just ignore the require (since we're already a debugger). This option is compatible with Ruby's.

`-s | --server`
: Debug the program but listen for remote connections on the default port or port set up via the '`--port`' option. See also '`--wait`'.

`-w | --wait`
: Debug the program but stop waiting for a client connection first. This option automatically sets '`--server`' option.

`-x | --trace`

Turn on line tracing. Running `rdebug --trace` *`rubyscript.rb`* is much like running: `ruby -rtracer` *`rubyscript.rb`*

If all you want to do however is get a linetrace, `tracer`, not `rdebug`, may be faster:

```
$ time ruby -rtracer gcd.rb 34 21 > /dev/null

real 0m0.266s
user 0m0.008s
sys 0m0.000s
$ time rdebug –trace gcd.rb 34 21 > /dev/null

real 0m0.875s
user 0m0.448s
sys 0m0.056s
$
```

2.1.2 How to Set Default Command-Line Options

ruby-debug has many command-line options; it seems that some people want to set them differently from the our defaults. For example, some people may want '`--no-quit --no-control`' to be the default behavior. One could write a wrapper script or set a shell alias to handle this. ruby-debug has another way to do this as well. Before processing command options if the file `$HOME/.rdboptrc` is found it is loaded. If you want to set the defaults in some other way, you can put Ruby code here and set variable `options` which is an OpenStruct. For example here's how you'd set '`-no-quit`' and change the default control port to 5000.

```
# This file contains how you want the default options to ruby-debug
# to be set. Any Ruby code can be put here.
#
# debugger # Uncomment if you want to debug rdebug!
options.control = false
options.port = 5000
puts "rocky's rdboptrc run"
```

Here are the default values in `options`

```
#<OpenStruct server=false, client=false, frame_bind=false, cport=8990, tracing=false, nx=false, post_mort
```

2.2 Command files

A command file for ruby-debug is a file of lines that are ruby-debug commands. Comments (lines starting with `#`) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When you start ruby-debug, it automatically executes commands from its *init files*, normally called '`.rdebugrc`'.

On some configurations of ruby-debug, the init file may be known by a different name. In particular on MS-Windows (but not cygwin) '`rdebug.ini`' is used.

During startup, ruby-debug does the following:

1. Processes command line options and operands.

2. Reads the init file in your current directory, if any, and failing that the home directory. The home directory is the directory named in the `HOME` or `HOMEPATH` environment variable.

   Thus, you can have more than one init file, one generic in your home directory, and another, specific to the program you are debugging, in the directory where you invoke ruby-debug.
3. Reads command files specified by the '`--script`' option.

You can also request the execution of a command file with the `source` command, see Section 3.5.4 [Source], page 26.

2.3 Quitting the debugger

An interrupt (often *C-c*) does not exit from ruby-debug, but rather terminates the action of any ruby-debugcommand that is in progress and returns to ruby-debug command level. Inside a debugger command interpreter, use `quit` command (see Section 3.5 [Quitting the debugger], page 25).

There way to terminate the debugger is to use the `kill` command. This does more forceful `kill -9`. It can be used in cases where `quit` doesn't work.

2.4 Calling the debugger from inside your Ruby program

Running a program from the debugger adds a bit of overhead and slows down your program a little.

Furthermore, by necessity, debuggers change the operation of the program they are debugging. And this can lead to unexpected and unwanted differences. It has happened so often that the term "Heisenbugs" (see `http://en.wikipedia.org/wiki/Heisenbug`) was coined to describe the situation where the addition of the use of a debugger (among other possibilities) changes behavior of the program so that the bug doesn't manifest itself anymore.

There is another way to get into the debugger which adds no overhead or slowdown until you reach the point at which you want to start debugging. However here you must change the script and make an explicit call to the debugger. Because the debugger isn't involved before the first call, there is no overhead and the script will run at the same speed as if there were no debugger.

There are three parts to calling the debugger from inside the script, "requiring" the debugger code, telling the debugger to start tracking things and then making the call calling the debugger to stop.

To get the debugger class accessible from your Ruby program:

```
require 'rubygems'
require 'ruby-debug'
```

(It is very likely that you've already require'd rubygems. If so, you don't have to do that again.) These commands need to be done only once.

After `require 'ruby-debug'`, it's possible to set some of the debugger variables influence preferences. For example if you want to have `rdebug`run a `list` command every time it stops you set the variable `Debugger.settings[:autolist]`. see Section 5.1.3 [Debugger.settings], page 46 has a list of variable settings and the default values. Debugger settings can also be set in `.rdebugrc` as debugger commands. see Section 2.2 [Command Files], page 20

To tell the debugger to start tracking things:

```
Debugger.start
```

There is also a `Debugger.stop` to turn off debugger tracking. If speed is crucial, you may want to start and stop this around certain sections of code. Alternatively, instead of issuing an explicit `Debugger.stop` you can add a block to the `Debugger.start` and debugging is turned on for that block. If the block of code raises an uncaught exception that would cause the block to terminate, the `stop` will occur. See Section 1.4 [Debugger.start with a block], page 10.

And finally to enter the debugger:

```
debugger
```

As indicated above, when `debugger` is run a `.rdebugrc` profile is read if that file exists.

You may want to do enter the debugger at several points in the program where there is a problem you want to investigate. And since `debugger` is just a method call it's possible enclose it in a conditional expression, for example:

```
debugger if 'bar' == foo and 20 == iter_count
```

Although each step does a very specific thing which offers great flexibility, in order to make getting into the debugger easier the three steps have been rolled into one command:

```
require "ruby-debug/debugger"
```

3 ruby-debug Command Reference

3.1 Command Interfaces

There are several ways one can talk to `ruby-debug` and get results. The simplest way is via a command-line interface directly talking to the debugger. This is referred to below as a “Local Interface”. It’s also possible to run the debugger and set up a port by which some other process can connect and control the debug session. This is called a “Remote Interface”. When you want to gain access to a remote interface you need to run `ruby-debug` using a “Control Interface”. This interface might not be the same process as the process running the debugged program and might not even be running on the same computer.

Other front-ends may use one of these and build on top and provide other (richer) interfaces. Although many of the commands are available on all interfaces some are not. Most of the time in this manual when we talk about issuing commands describing the responses elicited, we’ll assume we are working with the local interface.

3.2 Command Syntax

Usually a command is put on a single line. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in `step 5`. You can also use the `step` command with no arguments. Some commands do not allow any arguments.

Multiple commands can be put on a line by separating each with a semicolon (`;`). You can disable the meaning of a semicolon to separate commands by escaping it with a backslash.

For example, if you have `autoeval` (Section 3.13.2 [Autoeval], page 38) set, you might want to enter the following code to compute the 5th Fibonacci number:

```
# Compute the 5 Fibonaci number
(rdb:1) set autoeval on
(rdb:1) fib1=0; fib2=1; 5.times {|temp| temp=fib1; fib1=fib2; fib2 += temp }
SyntaxError Exception: compile error
/usr/bin/irb:10: syntax error, unexpected $end, expecting '}'
 5.times {|temp| temp=fib1
                          ^
(rdb:1) fib1=0\; fib2=1\; 5.times {|temp| temp=fib1\; fib1=fib2\; fib2 += temp }
5
(rdb:1) fib2
fib2
8
```

You might also consider using the `irb` command, Section 3.7.3 [irb], page 28, and then you won’t have to escape semicolons.

A blank line as input (typing just `<RET>`) means to repeat the previous command.

In the “local” interface, the Ruby Readline module is used. It handles line editing and retrieval of previous commands. Up arrow, for example moves to the previous debugger command; down arrow moves to the next more recent command (provided you are not already at the last command). Command history is saved in file `.rdebug_hist`. A limit is put on the history size. You can see this with the `show history size` command. See Section 3.13.10 [History], page 40 for history parameters.

3.3 Command Output

In the command-line interface, when `ruby-debug` is waiting for input it presents a prompt of the form `(rdb:`*x*`)`. If debugging locally, *x* will be the thread number. Usual the main thread is 1, so often you'll see `(rdb:1)`. In the control interface though *x* will be `ctrl` and in post-mortem debugging `post-mortem`.

In the local interface, whenever `ruby-debug` gives an error message such as for an invalid command, or an invalid location position, it will generally preface the message with `***`. However if annotation mode is on that the message is put in a `begin-error` annotation and no `***` appears.

3.4 Getting help ('`help`')

Once inside `ruby-debug` you can always ask it for information on its commands, using the command `help`.

`help`
`h`
: You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of commands:

```
(rdb:1) help
ruby-debug help v0.10.4
Type 'help <command-name>' for help on a specific command

Available commands:
backtrace  delete   enable  help    next  quit     show    undisplay
break      disable  eval    info    p     reload   source  up
catch      display  exit    irb     pp    restart  step    var
condition  down     finish  list    ps    save     thread  where
continue   edit     frame   method  putl  set      trace
```

`help` *`command`*
: With a command name as `help` argument, ruby-debugdisplays short information on how to use that command.

```
(rdb:1) help list
ruby-debug help v0.10.4
l[ist] list forward
l[ist] - list backward
l[ist] = list current line
l[ist] nn-mm list given lines
* NOTE - to turn on autolist, use 'set autolist'
(rdb:1)
```

3.4.1 Help on Subcommands

A number of commands have many sub-parameters or *subcommands*. These include `info`, `set`, `show`, `enable` and `disable`.

When you ask for help for one of these commands, you will get help for all of the subcommands that that command offers. Sometimes you may want help that subcommand and to do this just follow the command with its subcommand name. For example `help set annotate` will just give help about the annotate command. Furthermore it will give longer help than the summary information that appears when you ask for help. You don't need to list the full subcommand name, but just enough of the letters to make that subcommand distinct from others will do. For example, `help set an` is the same as `help set annotate`.

Some examples follow.

```
(rdb:1) help info
Generic command for showing things about the program being debugged.
--
List of info subcommands:
--
info args -- Argument variables of current stack frame
info breakpoints -- Status of user-settable breakpoints
info catch -- Exceptions that can be caught in the current stack frame
info display -- Expressions to display when program stops
info file -- Info about a particular file read in
info files -- File names and timestamps of files read in
info global_variables -- Global variables
info instance_variables -- Instance variables of the current stack frame
info line -- Line number and file name of current position in source file
info locals -- Local variables of the current stack frame
info program -- Execution status of the program
info stack -- Backtrace of the stack
info thread -- List info about thread NUM
info threads -- information of currently-known threads
info variables -- Local and instance variables of the current stack frame
```

```
(rdb:1) help info breakpoints
Status of user-settable breakpoints.
Without argument, list info about all breakpoints.  With an
integer argument, list info on that breakpoint.
```

```
(rdb:1) help info br
Status of user-settable breakpoints.
Without argument, list info about all breakpoints.  With an
integer argument, list info on that breakpoint.
```

3.5 Controlling the debugger (‘quit’, ‘restart’, ‘interrupt’, ‘source’)

3.5.1 Quit (‘quit’)

`quit` [`unconditionally`]
`exit`
`q`

To exit ruby-debug, use the `quit` command (abbreviated `q`), or alias `exit`.

A simple `quit` tries to terminate all threads in effect.

Normally if you are in an interactive session, this command will prompt to ask if you really want to quit. If you don’t want any questions asked, enter the “unconditionally”.

3.5.2 Restart (‘`restart`’)

`restart`
`R`

Restart the program. This is is a re-exec - all debugger state is lost. If command arguments are passed those are used. Otherwise the last program arguments used in the last invocation are used.

In not all cases will you be able to restart the program. First, the program should have been invoked at the outset rather than having been called from inside your program or invoked as a result of post-mortem handling.

Also, since this relies on the the OS `exec` call, this command is available only if your OS supports that `exec`; OSX for example does not (yet).

3.5.3 Interrupt (‘`interrupt`’)

`interrupt`
`i` Interrupt the program. Useful if there are multiple threads running.

3.5.4 Running Debugger Commands (‘`source`’)

`source` *`filename`*

Execute the command file *filename*.

The lines in a command file are executed sequentially. They are not printed as they are executed. If there is an error, execution proceeds to the next command in the file. For information about command files that get run automatically on startup, see Section 2.2 [Command Files], page 20.

3.6 Executing expressions on stop (‘`display`’, ‘`undisplay`’)

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that ruby-debug evaluates a statement each time your program stops or the statement is shown in line tracing. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
(rdb:1) display n
1: n = 3
```

This display shows item numbers, expressions and their current values. If the expression is undefined or illegal the expression will be printed but no value will appear.

```
(rdb:1) display undefined_variable
2: undefined_variable =
(rdb:1) display 1/0
3: 1/0 =
```

Note: this command uses `to_s` to in expressions; for example an array `[1, 2]` will appear as `12`. For some datatypes like an Array, you may want to call the `inspect` method, for example `display ARGV.inspect` rather than `display ARGV`.

`display` *`expr`*

Add the expression *expr* to the list of expressions to display each time your program stops or a line is printed when linetracing is on (see Section 3.6 [DisplayCommands], page 26).

`display` Display the current values of the expressions on the list, just as is done when your program stops.

`undisplay` [*num*]
`delete display` *num*
Remove item number *num* from the list of expressions to display.

`info display`
Show all display expressions

`disable display` *dnums* . . .
Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

`enable display` *dnums* . . .
Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

3.7 Evaluating and Printing Expressions (‘p’, ‘pp’, ‘putl’, ‘ps’, ‘irb’)

One way to examine and change data in your script is with the `eval` command (abbreviated `p`). A similar command is `pp` which tries to pretty print the result. Finally `irb` is useful when you anticipate examining or changing a number of things, and prefer not to have to preface each command, but rather work as one does in `irb`.

3.7.1 Printing an expression (‘eval’, ‘p’)

`eval` *expr*
`p` *expr*

Use `eval` or `p` to evaluate a Ruby expression, *expr*, same as you would if you were in `irb`. If there are many expressions you want to look at, you may want to go into irb from the debugger.

```
(rdb:p) p n
3
(rdb:1) p "the value of n is #{n}"
"the value of n is 3"
(rdb:1)
```

3.7.2 Pretty-Printing an expression (‘pp’, ‘putl’, ‘ps’))

`pp` Evaluates and pretty-prints *expr*

```
(rdb:1) p $LOAD_PATH
["/home/rocky/lib/ruby", "/usr/lib/ruby/site_ruby/1.8", "/usr/lib/ruby/site_ruby/1.8/i586-lin
(rdb:1) pp $LOAD_PATH
["/home/rocky/lib/ruby",
 "/usr/lib/ruby/site_ruby/1.8",
 "/usr/lib/ruby/site_ruby/1.8/i586-linux",
 "/usr/lib/ruby/1.8"]
```

`putl` If the value you want to print is an array, sometimes a columnized list looks nicer:

```
(rdb:1) putl $LOAD_PATH
/home/rocky/lib/ruby                      /usr/lib/ruby/site_ruby/1.8
/usr/lib/ruby/site_ruby/1.8/i586-linux   /usr/lib/ruby/1.8
```

Note however that entries are sorted to run down first rather than across. So in the example above the second entry in the list is `/usr/lib/ruby/site_ruby/1.8/i586-linux` and the *third* entry is `/usr/lib/ruby/site_ruby/1.8`.

If the value is not an array `putl` will just call pretty-print.

`ps` Sometimes you may want to print the array not only columnized, but sorted as well. The list of debugger help commands appears this way, and so does the output of the `method` commands.

```
(rdb:1) ps Kernel.private_methods
Digest                   initialize                 y
Pathname                 initialize_copy
Rational                 location_of_caller
active_gem_with_options  method_added
alias_method             method_removed
append_features          method_undefined
attr                     module_function
attr_accessor            private
attr_reader              protected
attr_writer              public
class_variable_get       remove_class_variable
class_variable_set       remove_const
define_method            remove_instance_variable
extend_object            remove_method
extended                 singleton_method_added
gcd                      singleton_method_removed
gem_original_require     singleton_method_undefined
include                  timeout
included                 undef_method
```

If the value is not an array, `ps` will just call pretty-print. See also the `methods`.

3.7.3 Run irb (‘`irb`’)

`irb` Run an interactive ruby session (`irb`) with the bindings environment set to the state you are in the program.

When you leave irb and go back to the debugger command prompt we show again the file, line and text position of the program in the same way as when entered the debugger. If you issue a `list` without location information, the default location used is the current line rather than the position may have gotten updated via a prior `list` command.

```
triangle.rb:4
def triangle(n)
(rdb:1) list
[-1, 8] in /home/rocky/ruby/triangle.rb
   1  #!/usr/bin/env ruby
   2  # Compute the n'th triangle number - the hard way
   3  # triangle(n) == (n * (n+1)) / 2
=> 4  def triangle(n)
   5    tri = 0
   6    0.upto(n) do |i|
   7      tri += i
   8    end
```

```
irb
>> (0..6).inject{|sum, i| sum +=i}
=> 21
>> exit
triangle.rb:4
def triangle(n)
(rdb:1) list # Note we get the same line range as before going into irb
[-1, 8] in /home/rocky/ruby/triangle.rb
   1  #!/usr/bin/env ruby
   2  # Compute the n'th triangle number - the hard way
   3  # triangle(n) == (n * (n+1)) / 2
=> 4  def triangle(n)
   5    tri = 0
   6    0.upto(n) do |i|
   7      tri += i
   8    end
```

3.8 Printing Variables (‘var’, ‘method’)

`var const` *object*
: Show the constants of *object*. This is basically listing variables and their values in *object*`.constant`.

`var instance` *object*
: Show the instance variables of *object*. This is basically listing *object*`.instance_variables`.

`info instance_variables`
: Show instance_variables of `@self`

`info locals`
: Show local variables

`info globals`
: Show global variables

`info variables`
: Show local and instance variables of `@self`

`method instance` *object*
: Show methods of *object*. Basically this is the same as running `ps object.instance_methods(false)` on *object*.

`method iv` *object*
: Show method instance variables of *object*. Basically this is the same as running

```
obj.instance_variables.each do |v|
   puts "%s = %s\n" % [v, obj.instance_variable_get(v)]
end
```

on *object*.

`signature` *object*
: Show procedure signature of method *object*. *This command is available only if the nodewrap is installed.*

```
def mymethod(a, b=5, &bock)
end
```

```
(rdb:1) method sig mymethod
Mine#mymethod(a, b=5, &bock)
```

on *object*.

`method` *`class-or-module`*
: Show methods of the class or module, *class-or-module*. Basically this is the same as running `ps object.methods` on *class-or-module*. on *class-or-module*.

3.9 Examining Program Source Files (‘`list`’)

ruby-debug can print parts of your script’s source. When your script stops, ruby-debug spontaneously prints the line where it stopped and the text of that line. Likewise, when you select a stack frame (see Section 3.11.3 [Selection], page 32) ruby-debug prints the line where execution in that frame has stopped. Implicitly there is a default line location. Each time a list command is run that implicit location is updated, so that running several list commands in succession shows a contiguous block of program text.

You can print other portions of source files by giving an explicit position as a parameter to the list command.

If you use ruby-debug through its Emacs interface, you may prefer to use Emacs facilities to view source.

To print lines from a source file, use the `list` command (abbreviated `l`). By default, ten lines are printed. Fewer may appear if there fewer lines before or after the current line to center the listing around.

There are several ways to specify what part of the file you want to print. Here are the forms of the `list` command.

`list` *`line-number`*
`l` *`line-number`*
: Print lines centered around line number *line-number* in the current source file.

`list`
`l`
: Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see Section 3.11.1 [Frames], page 32), this prints lines centered around that line.

`list -`
`l -`
: Print lines just before the lines last printed.

`list` *`first-last`*
: Print lines between *first* and *last* inclusive.

`list =`
: Print lines centered around where the script is stopped.

Repeating a `list` command with `RET` discards the argument, so it is equivalent to typing just `list`. This is more useful than listing the same lines again. An exception is made for an argument of ‘`-`’; that argument is preserved in repetition so that each repetition moves up in the source file.

3.10 Editing Source files (‘edit’)

To edit the lines in a source file, use the `edit` command. The editing program of your choice is invoked with the current line set to the active line in the program. Alternatively, you can give a line specification to specify what part of the file you want to print if you want to see other parts of the program.

You can customize to use any editor you want by using the `EDITOR` environment variable. The only restriction is that your editor (say `ex`), recognizes the following command-line syntax:

```
ex +number file
```

The optional numeric value +*number* specifies the number of the line in the file where to start editing. For example, to configure ruby-debug to use the `vi` editor, you could use these commands with the `sh` shell:

```
EDITOR=/usr/bin/vi
export EDITOR
gdb ...
```

or in the `csh` shell,

```
setenv EDITOR /usr/bin/vi
gdb ...
```

`edit` [*line specification*]
: Edit line specification using the editor specified by the `EDITOR` environment variable.

3.11 Examining the Stack Frame (‘where’, ‘up’, ‘down’, ‘frame’)

When your script has stopped, one thing you’ll probably want to know is where it stopped and some idea of how it got there.

Each time your script performs a function or sends a message to a method, or enters a block, information about this action is saved. The frame stack then is this a history of the blocks that got you to the point that you are currently stopped at.[1]

One entry in call stack is *selected* by ruby-debug and many ruby-debugcommands refer implicitly to the selected block. In particular, whenever you ask ruby-debugto list lines without giving a line number or location the value is found in the selected frame. There are special ruby-debugcommands to select whichever frame you are interested in. See Section 3.11.3 [Selecting a frame], page 32.

When your program stops, ruby-debug automatically selects the currently executing frame and describes it briefly, similar to the `frame` command.

After switching frames, when you issue a `list` command without any position information, the position used is location in the frame that you just switched between, rather than a location that got updated via a prior `list` command.

[1] More accurately we should call this a “block stack”; but we’ll use the name that is more commonly used. And internally in Ruby, there is “FRAME” structure which is yet slightly different.

3.11.1 Stack frames

The block stack is divided up into contiguous pieces called *stack frames*, *frames*, or *blocks* for short; each frame/block has a scope associated with it; It contains a line number and the source-file name that the line refers. If the frame/block is the beginning of a method or function it also contains the function name.

When your script is started, the stack has only one frame, that of the function `main`. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

ruby-debug assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your script; they are assigned by ruby-debug to give you a way of designating stack frames in ruby-debug commands.

3.11.2 Backtraces (‘`where`’)

A backtrace is essentially the same as the call stack: a summary of how your script got where it is. It shows one line per frame, for many frames, starting with the place that you are stopped at (frame zero), followed by its caller (frame one), and on up the stack.

`where` Print the entire stack frame; `info stack` is an alias for this command. Each frame is numbered and can be referred to in the `frame` command; `up` and `down` add or subtract respectively to frame numbers shown. The position of the current frame is marked with `-->`.

```
(rdb:1) where
--> #0 Object.gcd(a#Fixnum, b#Fixnum) at line /tmp/gcd.rb:6
    #1 at line /tmp/gcd.rb:19
```

3.11.3 Selecting a frame (‘`up`’, ‘`down`’, ‘`frame`’)

Commands for listing source code in your script work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

`up` [*n*] Move *n* frames up the stack. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. Using a negative *n* is the same thing as issuing a `down` command of the absolute value of the *n*. Using zero for *n* does no frame adjustment, but since the current position is redisplayed, it may trigger a resynchronization if there is a front end also watching over things.

n defaults to one. You may abbreviate `up` as `u`.

`down` [*n*] Move *n* frames down the stack. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. Using a negative *n* is the same as issuing a `up` command of the absolute value of the *n*. Using zero for *n* does no frame adjustment, but since the current position is redisplayed, it may trigger a resynchronization if there is a front end also watching over things.

n defaults to one.

`frame` [*n*] [`thread` *`thread-num`*]
: The `frame` command allows you to move from one stack frame to another, and to print the stack frame you select. *n* is the the stack frame number or 0 if no frame number is given; `frame 0` then will always show the current and most recent stack frame.

  If a negative number is given, counting is from the other end of the stack frame, so `frame -1` shows the least-recent, outermost or most "main" stack frame.

  Without an argument, `frame` prints the current stack frame. Since the current position is redisplayed, it may trigger a resynchronization if there is a front end also watching over things.

  If a thread number is given then we set the context for evaluating expressions to that frame of that thread.

3.12 Stopping and Resuming Execution

One important use of a debugger is to stop your program *before* it terminates, so that if your script runs into trouble you can investigate and find out why. However should your script accidentally continue to termination, it can be arranged for ruby-debugto not to leave the debugger without your explicit instruction. That way, you can restart the program using the same command arguments.

Inside ruby-debug, your script may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a debugger command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution.

3.12.1 Breakpoints (‘`break`’, ‘`catch`’, ‘`delete`’)

A *breakpoint* makes your script stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your script stops.

You specify the place where your script should stop with the `break` command and its variants.

`ruby-debug` assigns a number to each breakpoint when you create it; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on your script until you enable it again.

`break`
: Set a breakpoint at the current line.

`break` *`linenum`*
: Set a breakpoint at line *linenum* in the current source file. The current source file is the last file whose source text was printed. The breakpoint will stop your script just before it executes any of the code on that line.

`break` *`filename`*`:`*`linenum`*
: Set a breakpoint at line *linenum* in source file *filename*.

What may be a little tricky when specifying the filename is getting the name recognized by the debugger. If you get a message the message "`No source file named ...`", then you may need to qualify the name more fully. To see what files are loaded you can use the `info files` or `info file` commands. If you want the name `rdebug` thinks of as the current file, use `info line`.

Here's an example:

```
$ rdebug ~/ruby/gcd.rb 3 5
/home/rocky/ruby/gcd.rb:4   # Note this is the file name
def gcd(a, b)
(rdb:1) break gcd.rb:6
*** No source file named gcd.rb
(rdb:1) info line
Line 4 of "/home/rocky/ruby/gcd.rb"
(rdb:1) break /home/rocky/ruby/gcd.rb:6
Breakpoint 1 file /home/rocky/ruby/gcd.rb, line 6
(rdb:1) break ~/ruby/gcd.rb:10  # tilde expansion also works
Breakpoint 2 file /home/rocky/ruby/gcd.rb, line 10
(rdb:1) info file gcd.rb
File gcd.rb is not cached
(rdb:1) info file /home/rocky/ruby/gcd.rb
File /home/rocky/ruby/gcd.rb
         19 lines
```

`break` *`class`*`:`*`method`*

Set a breakpoint in class *class* method *method*. You can also use a period `.` instead of a colon `:`. Note that two colons `::` are not used. Also note a class *must* be specified here. If the method you want to stop in is in the main class (i.e. the class that `self` belongs to at the start of the program), then use the name `Object`.

`catch` [*`exception`*] [`on | 1 | off | 0`]

Set catchpoint to an exception. Without an exception name show catchpoints.

With an "on" or "off" parameter, turn handling the exception on or off. To delete all exceptions type "catch off".

`delete` [*`breakpoints`*]

Delete the breakpoints specified as arguments.

If no argument is specified, delete all breakpoints (ruby-debugasks confirmation. You can abbreviate this command as `del`.

`info breakpoints` [*`n`*]
`info break` [*`n`*]

Print a table of all breakpoints set and not deleted, with the following columns for each breakpoint:

Breakpoint Numbers ('`Num`'*)*
Enabled or Disabled ('`Enb`'*)*

Enabled breakpoints are marked with '`1`'. '`0`' marks breakpoints that are disabled (not enabled).

File and Line (‘`file:line`’)
: The filename and line number inside that file where of breakpoint in the script. The file and line are separated with a colon.

Condition
: A condition (an arithmetic expression) which when true causes the breakpoint to take effect.

If a breakpoint is conditional, `info break` shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that.

`info break` with a breakpoint number *n* as argument lists only that breakpoint.

Examples:

```
(rdb:1) info break
Breakpoints at following places:
Num  Enb What
1    y   gcd.rb:3
2    y   gcb.rb:28 if n > 1
(rdb:1) info break 2
2    y   gcb.rb:28 if n > 1
```

3.12.2 Disabling breakpoints (‘`disable`’, ‘`enable`’)

Rather than deleting a breakpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.

You disable and enable breakpoints and catchpoints with the `enable` and `disable` commands, optionally specifying one or more breakpoint numbers as arguments. Use `info break` to print a list of breakpoints and catchpoints if you do not know which numbers to use.

A breakpoint or catchpoint can have any different states of enablement:

- Enabled. The breakpoint stops your program. A breakpoint set with the `break` command starts out in this state.
- Disabled. The breakpoint has no effect on your program.

You can use the following commands to enable or disable breakpoints and catchpoints:

`disable` *`breakpoints`*
: Disable the specified breakpoints—or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate `disable` as `dis`.

`enable` *`breakpoints`*
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

Breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 3.12.4 [Resuming Execution], page 36.)

3.12.3 Break conditions (‘`condition`’)

The simplest sort of breakpoint breaks every time your script reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Ruby expression.

Break conditions can be specified when a breakpoint is set, by using ‘`if`’ in the arguments to the `break` command. A breakpoint with a condition evaluates the expression each time your script reaches it, and your script stops only if the condition is *true*. They can also be changed at any time with the `condition` command.

`condition` *`bnum expression`*
: Specify *expression* as the break condition for breakpoint *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (nonzero).

`condition` *`bnum`*
: Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

The debugger does not actually evaluate *expression* at the time the `condition` command (or a command that sets a breakpoint with a condition, like `break if ...`) is given, however.

Examples;

```
condition 1 x>5   # Stop on breakpoint 0 only if x>5 is true.
condition 1       # Change that! Unconditionally stop on breakpoint 1.
```

3.12.4 Resuming Execution (‘`step`’, ‘`next`’, ‘`finish`’, ‘`continue`’)

A typical technique for using stepping is to set a breakpoint (see Section 3.12.1 [Breakpoints], page 33) at the beginning of the function or the section of your script where a problem is believed to lie, run your script until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

Continuing means resuming program execution until your script completes normally. In contrast, *stepping* means executing just one more “step” of your script, where “step” may mean either one line of source code. Either when continuing or when stepping, your script may stop even sooner, due to a breakpoint or a signal.

3.12.4.1 Step (‘`step`’)

`step` [+-] [*`count`*]
: Continue running your program until the next logical stopping point and return control to ruby-debug. This command is abbreviated `s`.

  Like, the programming Lisp, Ruby tends implemented in a highly expression-oriented manner. Therefore things that in other languages that may appear to be a single statement are implemented in Ruby as several expressions. For example, in an “if” statement or looping statements a stop is made after the expression is evaluated but before the test on the expression is made.

  So it is common that a lines in the program will have several stopping points where in other debuggers of other languages there would be only one. Or you may have several statements listed on a line.

When stepping it is not uncommon to want to go to a different line on each step. If you want to make sure that on a step you go to a *different* position, add a plus sign ('`+`').

Note: step+ with a number count is not the same as issuing count step+ commands. Instead it uses count-1 step commands followed by a step+ command. For example, `step+ 3` *is the same as* `step; step; step+`*, not* `step+; step+; step+`

If you find yourself generally wanting to use `step+` rather than `step`, you may want to consider using `set forcestep`, (see Section 3.13.8 [Forcestep], page 39).

If you have `forcestep` set on but want to temporarily disable it for the next step command, append a minus, or `step-`.

With a count, `step` will continue running as normal, but do so *count* times. If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

3.12.4.2 Next ('`next`')

`next` [+] [*count*]

This is similar to `step`, but function or method calls that appear within the line of code are executed without stopping. As with step, if you want to make sure that on a step you go to a *different* position, add a plus sign ('`+`'). Similarly, appending a minus disables a `forcestep` temporarily, and an argument *count* is a repeat count, as for `step`.

3.12.4.3 Finish ('`finish`')

`finish` [*frame-number*]

Execute until selected stack frame returns. If no frame number is given, we run until the currently selected frame returns. The currently selected frame starts out the most-recent frame or 0 if no frame positioning (e.g `up`, `down` or `frame`) has been performed. If a frame number is given we run until *frame* frames returns.

If you want instead to terminate the program and debugger entirely, use `quit` (see Section 2.3 [Quitting the debugger], page 21).

Note: Releases before Ruby version 1.8.7 show the return line as the first line of the method. Starting with version 1.8.7, the last line executed will be shown as the return line. http://rubyforge.org/tracker/?func=detail&atid=22040&aid=18749&group_id=426

3.12.4.4 Continue ('`continue`')

`continue` [*line-specification*]
`c` [*line-specification*]

Resume program execution, at the address where your script last stopped; any breakpoints set at that address are bypassed.

The optional argument *line-specification* allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.

Should the program stop before that breakpoint is reached, for example, perhaps another breakpoint is reached first, in a listing of the breakpoints you won't see this entry in the list of breakpoints.

3.13 ruby-debug settings (`'set args'`, `'set autoeval'`..)

You can alter the way ruby-debug interacts with you using `set` commands.

The various parameters to `set` are given below. Each parameter name needs to to be only enough to make it unique. For example `set force` is a suitable abbreviation for `set forcestep`. The letter case is not important, so `set FORCE` or `set Force` are also suitable abbreviations.

Many `set` commands are either "on" or "off", and you can indicate which way you want set by supplying the corresponding word. The number 1 can be used for "on" and 0 for "off". If none of these is given, we will assume "on". A deprecated way of turning something off is by prefacing it with "no".

Each `set` command has a corresponding `show` command which allows you to see the current value.

3.13.1 Set/Show args

`set args` [*parameters*]
: Specify the arguments to be used if your program is rerun. If `set args` has no arguments, `restart` executes your program with no arguments. Once you have run your program with arguments, using `set args` before the next `restart` is the only way to run it again without arguments.

`show args`
: Show the arguments to give your program when it is started.

3.13.2 Set/Show auto-eval

`set autoeval` [`on` | `1` | `off` | `0`]
: Specify that debugger input that isn't recognized as a command should be passed to Ruby for evaluation (using the current debugged program namespace). Note however that we *first* check input to see if it is a debugger command and *only* if it is not do we consider it as Ruby code. This means for example that if you have variable called `n` and you want to see its value, you could use `p n`, because just entering `n` will be interpreted as the debugger "next" command.

See also Section 3.7.3 [irb], page 28 and Section 3.13.4 [Autoirb], page 39.

When autoeval is set on, you'll get a different error message when you invalid commands are encountered. Here's a session fragment to show the difference

```
(rdb:1) stepp
Unknown command
(rdb:1) set autoeval on
autoeval is on.
(rdb:1) stepp
NameError Exception: undefined local variable or method `stepp' for ...
```

`show args`
: Shows whether Ruby evaluation of debugger input should occur or not.

3.13.3 Execute "list" command on every breakpoint

3.13.4 Set/Show auto-irb

`set autoirb` [`on | 1 | off | 0`]
: When your program stops, normally you go into a debugger command loop looking for debugger commands. If instead you would like to directly go into an irb shell, set this on. See also Section 3.13.2 [Autoeval], page 38 or Section 3.7.3 [irb], page 28 if you tend to use debugger commands but still want Ruby evaluation occasionally.

`show autoirb`
: Shows whether the debugger will go into irb on stop or not.

3.13.5 Set/Show auto-reload

Set this on if the debugger should check to see if the source has changed since the last time it reread in the file if it has.

3.13.6 Set/Show basename

`set basename` [`on | 1 | off | 0`]
: Source filenames are shown as the shorter "basename" only. (Directory paths are omitted). This is useful in running the regression tests and may useful in showing debugger examples as in this text. You may also just want less verbose filename display.

  By default filenames are shown as with their full path.

`show basename`
: Shows the whether filename display shows just the file basename or not.

3.13.7 Set/Show call style

Sets how you want call parameters displayed; `short` shows just the parameter names; `tracked` is the most accurate but this adds overhead. On every call, scalar values of the parameters get saved. For non-scalar values the class is saved.

3.13.8 Set/Show Forces Different Line Step/Next

`set forcestep` [`on | 1 | off | 0`]
: Due to the interpretive, expression-oriented nature of the Ruby Language and implementation, each line often contains many possible stopping points, while in a debugger it is often desired to treat each line as an individual stepping unit.

  Setting forcestep on will cause each `step` or `next` command to stop at a different line number. See also Section 3.12.4.1 [Step], page 36 and Section 3.12.4.2 [Next], page 37.

`show forcestep`
: Shows whether forcestep is in effect or not.

3.13.9 Set/Show Frame full path

3.13.10 Command History Parameters

`show commands`
: Display the last ten commands in the command history.

`show commands` *`n`*
: Print ten commands centered on command number *n*.

`show history filename`
: Show the filename in which to record the command history (the list of previous commands of which a record is kept).

`set history save` [`on | 1 | off | 0`]
: Set whether to save the history on exit.

`show history save`
: Show saving of the history record on exit.

`set history size` *`number`*
: Set the maximum number of commands to save in the history.

`show history size`
: Show the size of the command history, i.e. the number of previous commands to keep a record of.

3.13.11 Save frame binding on each call

3.13.12 Set/Show Line tracing

`set linetrace` [`on | 1 | off | 0`]
: Setting linetrace on will cause lines to be shown before run.

`show linetrace`
: Shows whether line tracing is in effect or not.

3.13.13 Set/Show Line tracing style

`set linetrace+` [`on | 1 | off | 0`]
: Setting linetrace+ on will cause consecutive trace lines not to be a duplicate of the preceding line-trace line. Note however that this setting doesn't by itself turn on or off line tracing.

`show linetrace`
: Shows whether the line tracing style is to show all lines or remove duplicates linetrace lines when it is a repeat of the previous line.

3.13.14 Set/Show lines in a List command

`set listsize` *`number-of-lines`*
: Set number of lines to try to show in a `list` command.

`show listsize`
: Shows the list-size setting.

3.13.15 Show Post-mortem handling

Shows wither post-mortem debugging is in effect. Right now we don't have the ability to change the state inside the debugger.

3.13.16 Display stack trace when 'eval' raises exception

3.13.17 Set/Show Line width

`set width` *`column-width`*
: Set number of characters the debugger thinks are in a line. We also change OS environment variable `COLUMNS`.

`show width`
: Shows the current width setting.

3.14 Program Information ('`info`')

This `info` command (abbreviated `i`) is for describing the state of your program. For example, you can list the current parameters with `info args`, or list the breakpoints you have set with `info breakpoints` or `info watchpoints`. You can get a complete list of the `info` sub-commands with `help info`.

`info args` Method arguments of the current stack frame.

`info breakpoints`
: Status of user-settable breakpoints

`info display`
: All display expressions.

`info files`
: Source files in the program.

`info file` *`filename`* [*`all`*|*`lines`*|*`mtime`*|*`sha1`*]
: Information about a specific file. Parameter `lines` gives the number of lines in the file, `mtime` shows the modification time of the file (if available), `sha1` computes a SHA1 has of the data of the file. `all` gives all of the above information.

`info line` Line number and file name of current position in source.

`info locals`
: Local variables of the current stack frame.

`info program`
: Display information about the status of your program: whether it is running or not and why it stopped. If an unhandled exception occurred, the exception class and `to_s` method is called.

`info stack`
: Backtrace of the stack. An alias for `where`. See Section 3.11.2 [Backtrace], page 32.

`info thread` [*`thread-number`*] [`terse | verbose`]
: If no thread number is given, we list info for all threads. `terse` and `verbose` options are possible. If terse, just give summary thread name information. See

information under `info threads` for more detail about this summary information.

If `verbose` is appended to the end of the command, then the entire stack trace is given for each thread.

`info threads`

List information about currently-known threads. This information includes whether the thread is current (`+`), if it is suspended ($), or ignored (!); the thread number and the top stack item. If `verbose` is given then the entire stack frame is shown. Here is an example:

```
(rdb:7) info threads
  1 #<Thread:0xb7d08704 sleep> ./test/thread1.rb:27
 !2 #<Debugger::DebugThread:0xb7782e4c sleep>
  3 #<Thread:0xb777e220 sleep> ./test/thread1.rb:11
  4 #<Thread:0xb777e144 sleep> ./test/thread1.rb:11
  5 #<Thread:0xb777e07c sleep> ./test/thread1.rb:11
  6 #<Thread:0xb777dfb4 sleep> ./test/thread1.rb:11
+ 7 #<Thread:0xb777deec run> ./test/thread1.rb:14
(rdb:1)
```

Thread 7 is the current thread since it has a plus sign in front. Thread 2 is ignored since it has a `!`. A "verbose" listing of the above:

```
(rdb:7) info threads verbose
  1 #<Thread:0xb7d08704 sleep>
#0 Integer.join at line test/thread1.rb:27
#1 at line test/thread1.rb:27
 !2 #<Debugger::DebugThread:0xb7782e4c sleep>
  3 #<Thread:0xb777e220 sleep>
#0 sleep(count#Fixnum) at line test/thread1.rb:11
#1 Object.fn(count#Fixnum, i#Fixnum) at line test/thread1.rb:11
#2 at line test/thread1.rb:23
  4 #<Thread:0xb777e144 sleep>
#0 sleep(count#Fixnum) at line test/thread1.rb:11
#1 Object.fn(count#Fixnum, i#Fixnum) at line test/thread1.rb:11
#2 at line test/thread1.rb:23
  5 #<Thread:0xb777e07c sleep>
#0 sleep(count#Fixnum) at line test/thread1.rb:11
#1 Object.fn(count#Fixnum, i#Fixnum) at line test/thread1.rb:11
#2 at line test/thread1.rb:23
  6 #<Thread:0xb777dfb4 sleep>
#0 sleep(count#Fixnum) at line test/thread1.rb:11
#1 Object.fn(count#Fixnum, i#Fixnum) at line test/thread1.rb:11
#2 at line test/thread1.rb:23
+ 7 #<Thread:0xb777deec run>
#0 Object.fn(count#Fixnum, i#Fixnum) at line test/thread1.rb:14
#1 at line test/thread1.rb:23
```

`info variables`

Local and instance variables.

4 Post-Mortem Debugging

It is also to possible enter the debugger when you have an uncaught exception that is about to terminate our program. This is called *post-mortem debugging*. In this state many, of the debugger commands for examining variables and moving around in the stack still work. However some commands, such as those which imply a continuation of running code, no longer work.

The most reliable way to set up post-mortem debugging is to use the '`--post-mortem`' option in invoking `rdebug`. See Section 2.1.1 [rdebug command-line options], page 17. This traps/wraps at the debugger "load" of your Ruby script. When this is done, your program is stopped after the exception takes place, but before the stack has been unraveled. (Alas, it would be nice to if one could allow resetting the exception and continuing, but details of code in Ruby 1.8's `eval.c` prevent this.)

If however you haven't invoked `rdebug` at the outset, but instead call `ruby-debug` from inside your program, to set up post-mortem debugging set the `post_mortem` key in `Debugger.start`. Here's an example modified from `http://www.datanoise.com/articles/2006/12/20/post-mortem-debugging`:

```
$ cat t.rb
require 'rubygems'
require 'ruby-debug' ; Debugger.start(:post_mortem => true)

def t1
  raise 'test'
end
def t2
  t1
end
t2

$ ruby t.rb
t.rb:8: raise 'test'
(rdb:post-mortem) l=
[3, 12] in t.rb
   3
   4  Debugger.start
   5  Debugger.post_mortem
   6
   7  def t1
=> 8    raise 'test'
   9  end
   10  def t2
   11    t1
   12  end
(rdb:post-mortem)
```

Alternatively you can call `Debugger.post_mortem()` after rdebug has been started. The `post_mortem()` method can be called in two ways. Called without a block, it installs a global `at_exit()` hook that intercepts exceptions not handled by your Ruby script. In contrast to using the '`--post-mortem`' option, when this hook occurs after the call stack has been rolled back. (I'm not sure if this in fact makes any difference operationally; I'm just stating it because that's how it works.)

If you know that a particular block of code raises an exception you can enable post-mortem mode by wrapping this block inside a `Debugger.post_mortem` block

```
def offender
  1/0
end
...
require "ruby-gems"
require "ruby-debug"
Debugger.post_mortem do
  ...
  offender
  ...
end
```

Once inside the debugger in post-mortem debugging, the prompt should be `(rdb:post-mortem)`.

5 The Debugger Module and Class

5.1 The Debugger Module

5.1.1 `Debugger.start`, `Debugger.started?`, `Debugger.stop`, `Debugger.run_script`

In order to provide better debugging information regarding the stack frame(s) across all threads, ruby-debug has to intercept each call, save some information and on return remove it. Possibly, in Ruby 1.9 possibly this will not be needed. Therefore one has to issue call to indicate start saving information and another call to stop. Of course, If you call ruby-debug from the outset via `rdebug` this is done for you.

`Debugger.start([`*`options`*`])` [*`block`*]

Turn on add additional instrumentation code to facilitate debugging. A system even table hook is installed and some variables are set up to access thread frames.

This needs to be done before entering the debugger; therefore a call to the debugger issue a `Debugger.start` call if necessary.

If called without a block, `Debugger.start` returns `true` if the debugger was already started. But if you want to know if the debugger has already been started `Debugger.started?` can tell you.

If a block is given, the debugger is started and `yields` to block. When the block is finished executing, the debugger stopped with the `Debugger.stop method`. You will probably want to put a call to `debugger` somwhere inside that block

But if you want to completely stop debugger, you must call `Debugger.stop` as many times as you called Debugger.start method.

The first time Debugger.start is called there is also some additional setup to make the `restart` command work. In particular, `$0` and `ARGV` are used to set internal debugger variables.

Therefore you should make try to make sure that when `Debugger.start` is called neither of these variables has been modified. If instead you don't want this behavior you can pass an options has and set the `:init` key to `false`. That is

```
Debugger.start(:init => false) # or Debugger.start({:init => false})
```

If you want post-mortem debugging, you can also supply `:post_mortem => true` in `Debugger.start`.

`Debugger.started?`

Boolean. Return `true` if debugger has been started.

`Debugger.stop`

Turn off instrumentation to allow debugging. Return `true` is returned if the debugger is disabled, otherwise it returns `false`. *Note that if you want to stop debugger, you must call Debugger.stop as many times as you called the* `Debugger.start` *method.*

`Debugger.run_script(`*`debugger-command-file`*`, out = handler.interface)`
: Reads/runs the given file containing debugger commands. `.rdebugrc` is run this way.

`Debugger.last_exception`
: If not `nil`, this contains `$!` from the last exception.

5.1.2 `Debugger.context`

As mentioned previously, `Debugger.start` instruments additional information to be obtained about the current block/frame stack. Here we describe these additional `Debugger.context` methods.

Were a frame position is indicated, it is optional. The top or current frame position (position zero) is used if none is given.

`Debugger.context.frame_args` [*`frame-position=0`*]
: If track_frame_args? is true, return information saved about call arguments (if any saved) for the given frame position.

`Debugger.context.frame_args_info` [*`frame-position=0`*]
`Debugger.context.frame_class` [*`frame-position=0`*]
: Return the class of the current frame stack.

`Debugger.context.frame_file` [*`frame-position=0`*]
: Return the filename of the location of the indicated frame position.

`Debugger.context.frame_id` [*`frame-position=0`*]
: Same as `Debugger.context.method`.

`Debugger.context.frame_line` [*`frame-position=0`*]
: Return the filename of the location of the indicated frame position.

`Debugger.context.frame_method` [*`frame-position=0`*]
: Symbol of the method name of the indicated frame position.

`Debugger.context.stack_size`
: Return the number the size of the frame stack. Note this may be less that the actual frame stack size if debugger recording (`Debugger.start`) was turned on at after some blocks were added and not finished when the `Debugger.start` was issued.

5.1.3 `Debugger.settings`

Symbols listed here are keys into the Array `Debugger.settings`. These can be set any time after the `ruby-debug` is loaded. For example:

```
require "ruby-debug/debugger"
Debugger.settings[:autoeval] = true  # try eval on unknown debugger commands
Debugger.listsize = 20  # Show 20 lines in a list command
```

`:argv`
: Array of String. `argv[0]` is the debugged program name and `argv[1..-1]` are the command arguments to it.

`:autoeval`
: Boolean. True if auto autoeval on. See Section 3.13.2 [Autoeval], page 38.

`:autoirb` Fixnum: 1 if on or 0 if off. See Section 3.13.4 [Autoirb], page 39.

`:autolist`
: Fixnum: 1 if on or 0 if off.

`:basename`
: Boolean. True if basename on. See Section 3.13.6 [Basename], page 39.

`:callstyle`
: Symbol: `:short` or `:last`. See Section 3.13.7 [Callstyle], page 39.

`:debuggertesting`
: Boolean. True if currently testing the debugger.

`:force_stepping`
: Boolean. True if stepping should go to a line different from the last step. See Section 3.13.8 [Forcestep], page 39.

`:full_path`
: Boolean. See Section 3.13.9 [Fullpath], page 39.

`:listsize`
: Fixnum. Number of lines to show in a `list` command. See Section 3.13.14 [Listsize], page 40.

`:reload_source_on_change`
: Boolean. True if we should reread the source every time it changes. See Section 3.13.5 [Autoreload], page 39.

`:stack_trace_on_error`
: Boolean. True if we should produce a stack trace on error. See Section 3.13.16 [Trace], page 41.

`:width` Fixnum. Number of characters the debugger thinks are in a line. See Section 3.13.17 [Width], page 41.

5.2 The `Debugger` Class

`add_breakpoint(file, line, expr)`
: Adds a breakpoint in file *file*, at line *line*. If *expr* is not nil, it is evaluated and a breakpoint takes effect at the indicated position when that expression is true. You should verify that *expr* is syntactically valid or a `SyntaxError` exception, and unless your code handles this the debugged program may terminate.

`remove_breakpoint(bpnum)`
: When a breakpoint is added, it is assigned a number as a way to uniquely identify it. (There can be more than one breakpoint on a given line.) To remove a breakpoint, use `remove_breakpoint` with breakpoint number *bpnum*.

`breakpoints`
: Return a list of the breakpoints that have been added but not removed.

5.2.1 The `Debugger::Breakpoint` Class

Breakpoint are objects in the `Debugger::Breakpoint` class.

`enabled?` Returns whether breakpoint is enabled or not.

`enabled=` Sets whether breakpoint is enabled or not.

`expr` Expression which has to be true at the point where the breakpoint is set before we stop.

`expr=`

`hit_condition`
`hit_condition=`
`hit_count`
Returns the hit count of the breakpoint.

`hit_value`
Returns the hit value of the breakpoint.

`hit_value=`
Sets the hit value of the breakpoint.

`id` A numeric name for the breakpoint which is used in listing breakpoints and removing, enabling or disabling the breakpoint

`pos` Returns the line number of this breakpoint.

`pos=` Sets the line number of this breakpoint.

`source` Returns the file name in which the breakpoint occurs.

`source=` Sets the file name in which the breakpoint occurs.

5.2.2 The `Debugger::Context` Class

Callbacks in `Debugger:Context` get called when a stopping point or an event is reached. It has information about the suspended program which enable a debugger to inspect the frame stack, evaluate variables from the perspective of the debugged program, and contains information about the place the debugged program is stopped.

`at_line(`*`file`*`, `*`line`*`)`
This routine is called when the debugger encounters a “line” event for which it has been indicated we want to stop at, such as by hitting a breakpoint or by some sort of stepping.

`at_return(`*`file`*`, `*`line`*`)`
This routine is called when the debugger encounters a “return” event for which it has been indicated we want to stop at, such as by hitting a `finish` statement.

`debug_load(`*`file`*`, `*`stop-initially`*`)`
This method should be used to debug a file. If the file terminates normally, `nil` is returned. If not a backtrace is returned.

The *stop-initially* parameter indicates whether the program should stop after loading. If an explicit call to the debugger is in the debugged program, you may want to set this `false`.

5.2.3 The `Debugger::Command` Class

Each command you run is in fact its own class. Should you want to extend ruby-debug, it's pretty easy to do since after all ruby-debug is Ruby.

Each `Debugger#Command` class should have the a `regexp` method. This method returns regular expression for command-line strings that match your command. It's up to you to make sure this regular expression doesn't conflict with another one. If it does, it's undefined which one will get matched and run

In addition the instance needs these methods:

`execute`
: Code which gets run when you type a command (string) that matches the commands regular expression.

`help`
: A string which gets displayed when folks as for help on that command

`help_command`
: A name used the help system uses to show what commands are available.

Here's a small example of a new command:

```
module Debugger
  class MyCommand < Command
  def regexp
    /^\s*me$/ # Regexp that will match your command
  end

  def execute
    puts "hi" # What you want to happen when your command runs
  end
  class << self
    def help_command
      'me' # String name of command
    end
    def help(cmd)
   # Some sort of help text.
   %{This does whatever it is I want to do}
   end
  end
end
```

Now here's an example of how you can load/use it:

```
  require 'rubygems'
  require 'ruby-debug'
  require '/tmp/mycmd.rb' # or wherever
  Debugger.start
  x=1
  debugger
  y=2
```

And now an example of invoking it:

```
ruby /tmp/testit.rb:
/tmp/testit.rb:7
y=2
(rdb:1) help
ruby-debug help v0.10.3
Type 'help <command-name>' for help on a specific command
Available commands:
backtrace delete  enable help method putl    set    trace
```

```
break     disable eval   info next   quit    show   undisplay
catch     display exit   irb  p      reload  source up
condition down    finish list pp     restart step   var
continue edit     frame  me   ps     save    thread where
                  ^^ This is you

(rdb:1) help me
This does whatever it is I want to do
(rdb:1) me
hi
(rdb:1)
```

5.3 Additions to `Kernel`

`debugger` [*steps=1*]

Enters the debugger in the current thread after a stepping *steps* line-event steps. Before entering the debugger startup script is read.

Setting *steps* to 0 will cause a break in the debugger subroutine and not wait for eany line event to occur. This could be useful you want to stop right after the last statement in some scope.

Consider this example:

```
$ cat scope-test.rb

require 'rubygems'
require 'ruby-debug' ; Debugger.start
1.times do
   a = 1
   debugger   # implied steps=1
 end
y = 1

$ scope-test.rb:8
y = 1
(rdb:1) p a
NameError Exception: undefined local variable or method `a' for main:Object
(rdb:1)
```

The debugger will get at the line event which follows '`a=1`'. This is outside the `do` block scope where *a* is defined. If instead you want to stop before leaving the `do` loop it is possibly to stop right inside the `debugger`; call with 0 zero parameter:

```
$ cat scope-test.rb

require 'rubygems'
require 'ruby-debug' ; Debugger.start
1.times do
   a = 1
   debugger(0)
end
y = 1

$ scope-test.rb:8
../lib/ruby-debug-base.rb:175
Debugger.current_context.stop_frame = 0
(rdb:1) where
```

```
--> #0 Kernel.debugger(steps#Fixnum) at line ../lib/ruby-debug-base.rb:175
    #1 at line scope-test.rb:6
    #2 at line scope-test.rb:4
(rdb:1) up
#1 at line scope-test.rb:6
(rdb:1) p a
1
(rdb:1)
```

As seen above you will have to position the frame up one to be back in your debugged program rather than in the debugger.

`breakpoint` [*steps=1*]
: An alias for debugger.

`binding_n` [*n=0*]
: Returns a '`binding()`' for the *n*-th call frame. Note however that you need to first call '`Debugger.start`' before issuing this call.

Appendix A Building and Installing from rubyforge's Subversion Repository

Here are Unix-centric instructions. If you have Microsoft Windows or OSX some of the below may need adjusting.

A.1 Prerequisites: To build the package you'll need at a minimum:

- Ruby (of course). Currently only version 1.8.6 and above but not version 1.9.*x* work.
- Ruby development headers. This typically includes a file called '`ruby.h`'
- A C compiler like GNU C (`gcc`)
- Rake
- Subversion (`svn`).

If you want to build the documentation and install Emacs files, you'll also need:

- a POSIX shell like bash
- autoconf
- automake
- GNU Make
- texinfo

A.2 Basic Package Checkout and Installation

Check out the trunk of repository following the instructions at http://rubyforge.org/scm/?group_id=1900 For example on a Unixy system, this may work:

```
mkdir ruby-debug
cd ruby-debug
svn checkout svn://rubyforge.org/var/svn/ruby-debug/trunk trunk
```

In order to make the Ruby gems, `ruby-debug` and `ruby-debug-base`, get yourself into the trunk directory after the code has been checked out and run:

```
cd trunk # This is the same trunk checked out above.
rake package
```

If all goes well you should have some gem files put in the directory `pkg`. Use the gem command to install that.

```
sudo gem install ruby-debug-*.gem   # See gem help for other possibilities
```

If all goes well the rdebug script has been installed ruby-debug is now ready to run. But if everything goes well you might want to run the built-in regression tests to make sure everything is okay. See step 3 below.

If the gem install didn't work,'t there may be a problem with your C compiler or the Ruby headers are not installed.

A.3 Trying Out without Installing

You don't have to build a gem file to try out ruby debug. In fact when developing new features for ruby-debug, developers often you want to try it out *before* installing. If you have a problem in the latter part of step 1 you may want to try this approach since we go into a little more detail as to what happens under the covers when you do the gem install.

Run (from trunk)

```
rake lib
```

This creates a Makefile and builds the ruby-debug shared library. (On Unix the name is `ruby_debug.so`).

Once this is done you can run the debugger as you would rdebug using the script `runner.sh`. For example (again from trunk)

```
./runner.sh ~/my-ruby-program.rb
```

A.4 Running the Regression Tests

We've put together some basic tests to make sure ruby-debug is doing what we think it should do. To run these (from `trunk`):

```
rake test
```

If you didn't build the ruby-debug shared library and skipped step 2, don't worry `rake test` will do step 2 for you. You should see a line that ends something like:

```
Finished in 2.767579 seconds.

12 tests, 35 assertions, 0 failures, 0 errors
```

The number of seconds, tests, and assertions may be different from the above. However you *should* see exactly "0 failures, 0 errors."

A.5 Building the Documentation and Testing/Installing Emacs Files

Of course, I recommend you read the ruby-debug manual that comes with the package. If you have the prerequisites described above, run this once:

```
sh ./autogen.sh
```

Then run:

```
./configure
make
make test           # Runs Emacs regression tests
sudo make install   # Or arrange to do this as root
```

A.6 Building for Microsoft Windows

Microsoft Windows is "special" and building `ruby-debug-base` on it requires extra care. A problem here seems to be that the "One-click" install is compiled using Microsoft Visual Studio C, version 6 which is not sold anymore and is rather old.

Instead I suggest building via mingw/msys. `http://eigenclass.org/hiki.rb?cmd=view&p=cross+compil` has instructions on how to do. Some amendments to these instructions.

First, those instructions are a little GNU/Linux centric. If you are using Ubuntu or Debian, then this should be the easiest to follow the instructions. On Ubuntu or Debian there is a mingw3 Debian package. Installing that will give you the cross compiler that is a prerequisite. Alternatively if you are running MS Windows I notice that cygwin also has a mingw package. Or possibly you could use MinGW directly. For other OS's you might have to build a cross-compiler, i.e. gcc which emits win32 code and can create a win32 DLL.

After you have a cross compiler you need to download the Ruby source and basically build a ruby interpreter. The cross-compile.sh script works although when I downloaded it, it had lots of blank space at the beginning which will mess up the Unix magic interpretation. That is remove the blanks in front of `#/bin/sh`.

On my system, this script fails in running `make ruby` because the fake.rb that got created needed to have a small change:

```
ALT_SEPARATOR = "\"; \
```

should be:

```
ALT_SEPARATOR = "\\"; \
```

After fixing this, run `make ruby`. Also, I needed to run `make rubyw`.

And then `make install` as indicated.

Once all of that's in place, the place you want be is in `ruby-debug/trunk/ext/win32`, not `ruby-debug/ext`.

So let's say you've installed the cross-compiled install ruby in `/usr/local/ruby-mingw32/`. Here then are the commands to build `ruby-debug-base-`*xxx*`-mswin32.gem`:

```
cd .../ruby-debug/trunk/ext/win32
ruby -I /usr/local/ruby-mingw32/lib/ruby/1.8/i386-mingw32 ../extconf.rb
make # Not rake
cd ../..   # back in ruby-debug/trunk
rake win32_gem
```

Class, Module Method Index

Command Index

General Index

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt,
cmb10 at 10.95pt, and
cmsl10 at 10.95pt
are used for emphasis.