The Mindy Debugger

Copyright (c) 1994 Carnegie Mellon University All rights reserved. Refer to the end of this document for precise terms of use.

1. INTRODUCTION

When something goes wrong with your program, Mindy drops into the debugger. From the debugger, you can examine the stack, print out variables, evaluate expressions, and do various other things that can be helpful in figuring out what went wrong.

For example, if you did not define a method for main, after starting Mindy you would see something like the following:

```
No applicable methods for main with arguments #[]

thread [0] D   main
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

The first line is the error message. The second line tells you about the thread that encountered the error. For more information about threads see Section *Threads*. The third line tells you about the current stack frame for the thread; in this example, the last function called, which is at the top of the stack, is the **invoke-debugger** function. It was called with one argument, a **<simple-error>**.

The following sections discuss the various commands provided by the debugger. As a general rule, you can invoke a command by typing at least a unique prefix of its name. There are three commonly used commands for which a single letter suffices, regardless of all other command names:

- (d)own
- (l)ocals
- (c)ontinue

Throughout this document, some examples build on previous examples, even when those previous examples come from previous sections of the document. If there is a reference to the “previous example”, then please look to the previous section's text.

This documentation uses the term *built-in* for definitions created in C code, within Mindy's implementation. Opposed to built-in definitions are definitions written in Dylan code. Parts of Mindy are built-in and parts are defined in Dylan code. Of course, all user code is written in Dylan. Some debugger commands behave differently depending on whether the object being manipulated was defined in Dylan code or built-in.

2. Stack Manipulation Commands

The Mindy debugger offers a few commands for moving up and down the stack. The two most common commands are `up` and `down`. Mindy considers the most recently called function to be at the top of the stack and the least recently called function to be at the bottom of the stack. Hence, moving down the stack moves you from a callee to its caller. For example, if you were to type `down` after the previous example, you would see something like the following:

```
mindy> down
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #())
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
132     signal(cond);
mindy>
```

The first line tells you about the new current frame, which is a call to the **error** function. For a function written in Dylan, as opposed to a built-in function, the debugger tries to show the line of source code associated with the current frame. If the debugger could not find the source file, it still prints the line number from the source file.

While moving down the stack, you might have expected to see a call to the **signal** function before seeing a call to the **error** function. This does not happen because **signal** tail calls **invoke-debugger**. When a function tail calls another function, the callee reuses the current stack frame of the caller.

In addition to the `up` and `down` commands, you can move to a specified stack frame using the `frame` command. The debugger numbers stack frames starting at zero at the top of the stack. Currently, the debugger does not print frame numbers when it prints frame information, so moving with the `frame` command is only useful as a rough thumb bar. The following is an example of using this command to go to the top of the stack:

```
mindy> frame 0
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

If you use the `frame` command without supplying a frame number, the command prints the current frame's information. This is useful if the description of the current frame has scrolled off the screen, and you want to see it again.

You can view the entire stack by using the `backtrace` command. The current frame stays the same, but the `backtrace` command always shows the entire stack from the top to the bottom. The following is example output from this command:

```
mindy> backtrace
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #()) \
[/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan, line 132]
fp 0x10034058: main()
mindy>
```

3. Examining Variables

The `locals` command prints the value for every local variable in the function associated with the current frame. If you were at the frame for the **error** call in the previous example, using the `locals` command would look like the following:

```
mindy> locals
noise: #[]
cond: {<simple-error> 0x1023fa91}
mindy>
```

You can use the `print` command to print a specific local variable. The following is an example of printing the **cond** variable shown in the previous sample output:

```
mindy> print cond
$0={<simple-error> 0x1023fa91}
mindy>
```

The `print` command can also print the value of global variables:

```
mindy> print size
$1={<generic-function> size}
mindy>
```

For information on the labels the debugger assigns to values (that is, the `$N` identifications), see Section *Debugger Variables*.

If the debugger does not find a local variable with the name you supplied, the debugger looks for a global variable by that name in the current library and module. For more information about libraries and modules, see Section *Libraries and*

Modules. For more information about the `print` command, see Section *Evaluating Expressions*.

4. Libraries and Modules

When evaluating expressions, the debugger uses the *current library* and *current module*. When the debugger starts up, it guesses at what library and module to make current. If you want to access a global variable from another module or library, you first make another module or library be the current one with the `library` or `module` command. If you invoke the `library` command without an argument, it lists the available libraries and tells you which one is the current one. If you invoke the `library` command with an argument, the debugger makes that library be the current library. In the same way, the `module` command either lists the modules of the current library, or it selects another module be the current module. The following are examples of using the `library` command:

```
mindy> library
Dylan-User
Dylan

Current library is Dylan
mindy> library dylan-user
mindy>
```

The following is an example of using the `module` command after having just switched to the Dylan-user library:

```
mindy> module
   Dylan-User
 i File-Descriptors
 i Threads
 i Extensions
 i System
 i Dylan

The current module is Dylan-User
mindy>
```

The `i` in the second column indicates that those modules are being imported into the Dylan-user library as opposed to being defined there. The `module` command also indicates which modules are exported from the current library. For example, if you were to switch to the Dylan library, the `module` command would produce the following output:

```
mindy> library dylan
mindy> module
   Dylan-User
 x  File-Descriptors
 x  Threads
    Builtin-Stuff
 x  Extensions
 x  System
 x  Dylan

The current module is Dylan-User
mindy>
```

The `x` in the first column indicates that those modules are exported. There were no `x`s in the listing of modules in the Dylan-user library because no modules are exported from the Dylan-user library. There were no `i`s in the listing of modules for the Dylan library because the Dylan library does not import any modules.

Whenever you change libraries with the `library` command, the debugger resets the current module to the Dylan-user module. This is because the debugger needs to make a module current in the new library, and every library has a Dylan-

user module.

5. Evaluating Expressions

The `print` command can evaluate simple expressions and print their results. The following is an example:

```
mindy> print list(1, 2, 3)
$2=#(1, 2, 3)
mindy> print vector(4, 5, 6)
$3=#[4, 5, 6]
mindy>
```

The `print` command evaluates the variable `list` and then invokes that function with the arguments `1`, `2`, and `3`. The debugger labels values printed with a dollar sign and a number, and you can use these labels in later expressions. For more information on these, see Section *Debugger Variables*.

The expressions that the debugger accepts are limited. An expression can be one of the following:

- One of the following literals:
  - decimal number (`47`)
  - keyword (`foo:`)
  - string (`"foo"`)
  - `#t`
  - `#f`
- A variable name.
- A debugger variable (for example, `$5`).
- A function call (for example, `foo(a, b)` and `bar(c, quux: 3)`)
- The address, in hexidecimal (C format, not Dylan), of a valid dylan object (for example, `0x102050b1`). Note: use this feature with care, as a mistyped address can cause Mindy to dump core.

If the expression results in multiple values, all the values are printed on a single line:

```
mindy> print values(1, 2, 3)
$4=1, $5=2, $6=3
```

If an error occurs while the debugger is evaluating the expression, it prints the error message, aborts the `print` command, and returns to the debugger prompt. The following is an example of this situation:

```
mindy> print error("oops")
invocation failed:
  oops
mindy>
```

The `call` command is like the `print` command, but the `call` command does not handle errors by aborting. When you use the `call` command, and the expression causes an error, the debugger returns to its prompt, but any stack frames that were created due to the `call` command are now visible for inspection. The following is an example of using the `call` command:

```
mindy> call error("oops")

oops

thread [0] D   main
fp 0x100341f4: invoke-debugger({<simple-error> 0x102456b1})
mindy>
```

The `print` and `call` commands can also evaluate multiple, comma-separated expressions:

```
mindy> print 1, 2, 3
$7=1
$8=2
$9=3
mindy>
```

6. Debugger Variables

The `print` or `call` commands label every value printed, and these labels identify *debugger variables*. You can use these identifers in later expressions to refer to previously computed values. The following is an example:

```
mindy> p list(1, 2, 3)
$4=#(1, 2, 3)
mindy> p second($4)
$5=2
mindy>
```

The notation `$-N` provides a dynamic alternative to identifying debugger variables. This notation refers to previously printed values by using `N` as a count from the most recently printed value to the least recently printed. The counting begins at one.

```
mindy> print a:, b:, c:, d:
$12=a
$13=b
$14=c
$15=d
mindy> print $-1, $-2, $-3, $-4
$16=d
$17=c
$18=b
$19=a
mindy>
```

You can use `$` as a shorthand for `$-1`, and `$$` for `$-2`:

```
mindy> p 2
$20=2
mindy> p list($, 4)
$21=#(2, 4)
mindy> p list($$, 6)
$22=#(2, 6)
mindy>
```

Mindy keeps references to all debugger variables to prevent them from being garbage collected. If you no longer care about previously printed values, you might want to use the `flush` command to get rid of them:

```
mindy> flush
Flushed all debugger variables.
mindy> p $0
invocation failed:
No debug variable $0
mindy> p list(a:, b:, c:)
$0=#(a, b, c)
mindy>
```

You can use `$aN` notation to refer to the arguments passed to the function call associated with the current stack frame. `N` is the argument number, counting from zero. The following is an example:

```
mindy> frame
fp 0x10034078: error({<simple-error> 0x1023fa91}, #[], #())
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
132     signal(cond);
mindy> p $a0
$1={<simple-error> 0x1023fa91}
mindy> p $a1
$2=#[]
mindy> p $a2
$3=#()
mindy>
```

The `$aN` notation does not identify a debugger variable, and the debugger does not have to create storage for these values because they are already stored on the call stack. The `flush` command has no effect on argument values.

7. Restarts and Returning

This section discusses invoking Dylan restart handlers and returning values for conditions whose recovery protocols allow returning. If you do not know what these are, see the *Dylan Interim Reference Manual*.

The debugger has commands that allow you to try to continue executing your program. The most common way to continue execution is to invoke a Dylan restart. To either list the available restarts or invoke a restart, you use the `restart` command:

```
mindy> call cerror("go on", "oops")

oops

thread [0] D   main
fp 0x1003428c: invoke-debugger({<simple-error> 0x10245361})
mindy> restart
0 [{class <simple-restart>}]: go on
1 [{class <abort>}]: Blow off call
mindy> restart 0
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

In this example, the `restart` command lists two restarts. The **cerror** function establishes the `"go on"` restart (numbered 0). The `call` command establishes the `"Blow off call"` restart (numberd 1). The `restart 0` command caused **cerror** to return `#f`, which the `call` command printed.

The `abort` command invokes the first restart that handles **<abort>** restarts. The following is an example of this command:

```
mindy> call error("oops")

oops

thread [0] D   main
fp 0x100341fc: invoke-debugger({<simple-error> 0x10241d49})
mindy> abort
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

If Mindy entered the debugger due to a condition that allows returning as part of its recovery protocol, then you can use the `return` command. For example, consider an **<ignorable-error>** condition that is a subclass of **<error>** and that allows returning as part of its recovery protocol. The following example shows returning from the signalling of this condition:

```
mindy> call signal(make(<ignorable-error>))

{<ignorable-error> 0x10247759}

thread [0] D   main
fp 0x100341d4: invoke-debugger({<ignorable-error> 0x10247759})
mindy> restart
0 [{class <abort>}]: Blow off call

Returning is allowed:
  ignore it.
mindy> return
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10244831})
mindy>
```

8. Interrupting and Single Stepping

Sometimes it is useful to interrupt your program to see where it is currently executing. Consider the following program as an example:

```
module: Dylan-User

define method main (#rest noise)
  foo(#t);
end;

define method foo (x)
  if (x)
    foo(#f);
  else
    foo(#t);
  end;
end;
```

If you were to run this program and then interrupt it, you would see output similar to the following:

```
^C
Interrupted
thread [0] R   main
fp 0x10034060: foo(#f, #())
foo.dylan
8       if (x)
mindy>
```

After interrupting the program you have the full debugger at your disposal, as if an error had occurred. Additionally, you can use the `continue` command to resume execution:

```
mindy> continue
```

You can also use the `step` command to advance line by line through your program. When stepping, if the debugger encounters a function call, it descends into that function and steps line by line. The following is an example:

```
^C
Interrupted
thread [0] R   main
fp 0x10034060: foo(#f, #())
foo.dylan
8       if (x)
mindy> step
foo.dylan
11      foo(#t)
mindy> step
foo.dylan
8       if (x)
mindy> step
foo.dylan
9       foo(#f)
mindy> step
foo.dylan
8       if (x)
mindy>
```

9. Breakpoints

The debugger has a primitive facility for setting breakpoints in methods written in Dylan, as opposed to built-in methods. The `breakpoint` command takes two arguments, a reference to a method in which to install the breakpoint, and the line number at which to install the breakpoint. For example, consider the following program:

```
module: dylan-user

define constant foo =
  method ()
    puts("this is a test\n");
    puts("of breakpoints.\n");
    #f;
  end;
```

If you were to put a breakpoint at line 6 (the second `puts`), Mindy would produce output similar to the following:

```
mindy> break foo, 6
breakpoint 1 installed in {anonymous <byte-method> 0x10243d31\
#()} at line 6 (pc 47)
mindy> call foo()
this is a test
Breakpoint
thread [0] R   main
fp 0x100341dc: {anonymous <byte-method> 0x10243d31 #()}(#())
foo.dylan
6       puts("of breakpoints.\n");
mindy>
```

The `continue` and `step` commands can be used to continue execution (see Section *Interrupting and Single Stepping*):

```
mindy> step
of breakpoints.
foo.dylan
7       #f;
mindy> c
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10243e49})
mindy>
```

The `breakpoint` command evaluates its first argument, so you can use an arbitrary expression for the function. For example, you could use **find-method** to extract a specific method from a generic function and insert a breakpoint in that method:

```
mindy> br find-method(size, list(<table>)), 886
breakpoint 1 installed in {<byte-method> size #({class <table>})}\
at line 886 (pc 35)
mindy>
```

The `breakpoint` command with no arguments lists the currently installed breakpoints:

```
mindy> breakpoint
id  where
 1  pc 47 in {<component> 0x10204ea9}
mindy>
```

The `delete N` command removes a breakpoint, where `N` is the breakpoint ID reported in the `breakpoint` listing.

Sometimes the Mindy compiler has to split a single top level form into multiple methods. When this happens, the debugger cannot always figure out where to insert your breakpoint. Consider the following program:

```
module: dylan-user

define constant foo =
  method ()
    block (exit)
      puts("this is a test\n");
      puts("of breakpoints.\n");
      #f;
    end;
  end;
```

When this program is compiled, the compiler has to put the contents of the block in a seperate method. Because of this, if you were to try to insert a breakpoint at line 7 it would not work:

```
mindy> break foo, 7
{anonymous <byte-method> 0x10243f59 #()} does not span line n\
umber 7
mindy>
```

To insert a breakpoint into this method, you need to use the `disassemble` command. It disassembles a method and all Mindy-generated methods that might be associated with that method. For example:

```
mindy> disassemble foo
anonymous component, from "foo.dylan"
5           block (exit)
    47: b0              push    function catch
    48: 21              push    const(1)        {<method-info> 0x10205149}
    49: b2              push    function list
    50: a3              push    value <object>
    51: 91              call    nargs = 1, for single
    52: 0e              push    #()
    53: 10              push    #t
    54: 06              make-method
    55: 71              call    nargs = 1, tail
{<method-info> 0x10205149}, anonymous component, from "foo.dylan"
5           block (exit)
    51: 31              push    arg(1)
    52: 20              push    const(0)        {<method-info> 0x102050b1}
    53: b1              push    function list
    54: 90              call    nargs = 0, for single
    55: 0e              push    #()
    56: 10              push    #t
    57: 06              make-method
    58: 60              pop     local(0)
6             puts("this is a test\n");
    59: b2              push    function puts
    60: 23              push    const(3)        "this is a test\n"
    61: 81 00           call    nargs = 1, for 0
7             puts("of breakpoints.\n");
    63: b2              push    function puts
    64: 24              push    const(4)        "of breakpoints.\n"
    65: 81 00           call    nargs = 1, for 0
8             #f;
    67: 11              push    #f
    68: 02              return single

{<method-info> 0x102050b1}, exit component, from "foo.dylan"
5           block (exit)
    39: b0              push    function apply
    40: a1              push    value throw
    41: 30              push    arg(0)
    42: 32              push    arg(2)
    43: 73              call    nargs = 3, tail
    mindy>
```

As you can see, the function **foo** has been split into three methods. The first one corresponds to the part of **foo** that is outside the block. The second method corresponds to the code inside the block. And the third one corresponds to the **exit** function established by the block. Look for the second method which spans line 7. The following shows how to install the breakpoint:

```
mindy> br 0x10205149, 7
breakpoint 1 installed in {<method-info> 0x10205149} at line 7 (pc 63)
mindy> call foo()
this is a test
Breakpoint
thread [0] R   main
fp 0x100341f8: {anonymous <byte-method> 0x10245f41 #({class <object>})}\
({<catch> 0x10245f81}, #())
foo.dylan
7       puts("of breakpoints.\n");
mindy> c
of breakpoints.
$0=#f
fp 0x10034090: invoke-debugger({<simple-error> 0x10244071})
mindy>
```

10. Threads

Normally, there is only one thread of execution, in which case you won't need any of the commands in this section. When you debug a multi-threaded program, these commands become very useful. The `thread` command either lists the available threads or switches between them, depending on how you invoke it. For example:

```
mindy> p spawn-thread(foo:, curry(break, "Thread foo"))
$0={<thread> 0x10243f49}
mindy> p spawn-thread(bar:, curry(break, "Thread bar"))
$1={<thread> 0x10246f19}
mindy> thread
c [0] D   main
  [1] R   foo
  [2] R   bar
mindy>
```

In this example, the `thread` command lists three threads: the `main` (or original) thread and the two threads you just created. The `c` in the first column indicates which thread the debugger is currently examining. The `[N]` indicates the thread ID for each thread. The `D` and `R` designations indicate the status of each thread. The `main`, `foo`, and `bar` labels are the debug-names passed as the first argument to **spawn-thread**.

The different thread status codes are as follows:

```
STATUS     MEANING
   D       current thread the debugger is examining
   R       running/runable
   S       suspended
   B       blocked on a lock
   W       waiting for an event
```

Giving the `thread` command an argument causes the debugger to examine another thread. You can designate threads with either its numeric ID or the debug-name passed to **spawn-thread**:

```
mindy> thread foo
thread [1] R   foo
fp 0x102550bc: {anonymous <byte-method> 0x102443d9 #({class <\
object>})}({<catch> 0x10244421}, #(), {<value-cell> 0x1024436\
9}, {<breakpoint> 0x102441e1})
/afs/cs.cmu.edu/project/gwydion/mindy/src/runtime/cond.dylan
```

```
212     init-arguments: list(format-string: "Continue from break"))
mindy> thread 0
thread [0] D   main
fp 0x10034090: invoke-debugger({<simple-error> 0x1023fa91})
mindy>
```

Sometimes it is useful to temporarily disable some threads while debugging other threads. The `disable <thread-id-or-name>` command disables (suspends) the indicated thread, and the `enable` command allows a thread to run again:

```
mindy> disable foo
[1] S 1 foo
mindy> enable foo
[1] R   foo
mindy>
```

In this example, The status of the `foo` thread changes from `R` (runnable) to `S` (suspended) when it is disabled.

If you repeatedly use the `disable` command on the same thread, then the `enable` command must be used the same number of times to before the thread's status changes to `R`. The `1` after the `S` above is the number of times the thread `foo` has been disabled.

When a thread is suspended, the `continue` and `step` commands do not advance the thread's execution. The `disable` and `enable` commands can help you find thread synchronization problems by allowing you to explicitly control when each thread runs.

Invoking the `disable` or `enable` command with no argument affects the current thread the debugger is examining.

The `kill <thread-id-or-name>` command kills the indicated thread.

11. Miscellaneous Commands

The `help` command prints a one line summary of all the debugger commands.

The `quit` command causes Mindy to exit without executing any of the **on-exit** hooks. If you want the **on-exit** hooks to run, you should invoke the **exit** function with the `print` command:

```
mindy> print exit()
```

The `tron` command turns on an internal trace facility that prints the arguments and results for every function call. The `troff` command turns this off.

The `error` command repeats the error message for the condition that caused this thread to drop into the debugger.

The `gc` command invokes the garbage collector.

The `describe` command takes an expression as an argument and evaluates it. If the result is an instance of a class defined in Dylan, as opposed to a built-in class, then the debugger identifies the class, prints the slot names, and prints the slot values. If the result of the expression is an instance of a built-in class, then the debugger prints the value and its class. The `describe` command does not create or assign to debugger variables, but you can use debugger variables in the expression given to the command. The following examples show the `describe` command:

```
mindy> describe make
{<generic-function> make} is an instance of {class <generic-function>}

mindy> describe "Testing"
"Testing" is an instance of {class <byte-string>}
```

```
mindy> describe make(<table>)
{<object-table> 0x10245d79} is an instance of {class <object-table>}
and has the following slots:
merged-hash-state-slot: {permanent hash state}
shrink-to-slot: 100
shrink-when-slot: 10
expand-to-slot: 300
expand-when-slot: 200
bucket-states-slot: #[{permanent hash state}, {permanent hash state}, \
{permanent hash state}, {permanent hash state}, {permanent hash state}]
bucket-count-slot: 5
bucket-array-slot: #[#(), #(), #(), #(), #()]
item-count-slot: 0
```

12. Copyright and Terms Of Use

Copyright (c) 1994 Carnegie Mellon University All rights reserved.

Use and copying of this software and preparation of derivative works based on this software are permitted, including commercial use, provided that the following conditions are observed:

- This copyright notice must be retained in full on any copies and on appropriate parts of any derivative works.
- Documentation (paper or online) accompanying any system that incorporates this software, or any part of it, must acknowledge the contribution of the Gwydion Project at Carnegie Mellon University.

This software is made available *as is*. Neither the authors nor Carnegie Mellon University make any warranty about the software, its performance, or its conformity to any specification.

Bug reports, questions, comments, and suggestions should be sent by E-mail to the Internet address `gwydion-bugs@cs.cmu.edu`.