kshdb Documentation

Release 1.0.1

Rocky Bernstein

Nov 04, 2019

Contents:

kshdb is a gdb-like debugger for ksh.

Since this debugger is similar to other trepanning debuggers and *gdb* in general, knowledge gained by learning this is transferable to those debuggers and vice versa.

An Emacs interface is available via realgud.

CHAPTER 1

Features

Since this debugger is similar to other trepanning debuggers and *gdb* in general, knowledge gained by learning this is transferable to those debuggers and vice versa.

1.1 Source-code Syntax Colorization

Terminal source code is colorized via pygments . And with that you can set the pygments color style, e.g. "colorful", "paraiso-dark". See set_style . Furthermore, we make use of terminal bold and emphasized text in debugger output and help text. Of course, you can also turn this off.

1.2 Terminal Handling

We can adjust debugger output depending on the line width of your terminal. If it changes, or you want to adjust it, see set_width .

1.3 Smart Eval

If you want to evaluate the current source line before it is run in the code, use `eval`. To evaluate text of a common fragment of line, such as the expression part of an *if* statement, you can do that with `eval?`. See eval for more information.

1.4 More Stepping Control

Sometimes you want small steps, and sometimes large stepping.

This fundamental issue is handled in a couple ways:

1.4.1 Step Granularity

There are now `step` *event* and `next` *event* commands with aliases to `s+`, `s>` and so on. The plus-suffixed commands force a different line on a subsequent stop, the dash-suffixed commands don't. Without a suffix you get the default; this is set by the *set different* command.

CHAPTER 2

How to install

2.1 git

Many package managers have back-level versions of this debugger. The most recent versions is from the github.

To install from git:

```
$ git-clone git://github.com/rocky/kshdb.git
$ cd kshdb
$ ./autogen.sh  # Add configure options. See ./configure --help
```

If you've got a suitable *ksh* installed, then

```
$ make && make test
```

To try on a real program such as perhaps */etc/default/profile*:

```
$ ./kshdb -L /etc/default/profile # substitute .../profile with your favorite ksh␣
↪script
```

To modify source code to call the debugger inside the program:

```
source path-to-kshdb/kshdb/dbg-trace.sh
# work, work, work.

_Dbg_debugger
# start debugging here
```

Above, the directory *path-to-kshdb* should be replaced with the directory that *dbg-trace.sh* is located in. This can also be from the source code directory *kshdb* or from the directory *dbg-trace.sh* gets installed directory. The "source" command needs to be done only once somewhere in the code prior to using *_Dbg_debugger*.

If you are happy and *make test* above worked, install via:

```
sudo make install
```

and uninstall with:

```
$ sudo make uninstall # ;-)
```

CHAPTER 3

Entering the Ksh Debugger

Contents

3.1 Invoking the Debugger Initially

The simplest way to debug your program is to call run *kshdb*. Give the name of your program and its options and any debugger options:

```
$ cat /etc/profile

if [ "${PS1-}" ]; then
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
```

(continues on next page)

(continued from previous page)

```
  unset i
fi

$ kshdb /etc/profile
```

For help on *kshdb* or options, use the `--help` option.

```
$ kshdb --help

Usage:
   kshdb [OPTIONS] <script_file>

Runs ksh <script_file> under a debugger.

options:
...
```

3.2 Calling the debugger from your program

Sometimes it is not feasible to invoke the program from the debugger. Although the debugger tries to set things up to make it look like your program is called, sometimes the differences matter. Also the debugger adds overhead and slows down your program.

Another possibility then is to add statements into your program to call the debugger at the spot in the program you want. To do this, you source *kshdb/dbg-trace.sh* from where wherever it appears on your filesystem. This needs to be done only once.

After that you call *_Dbg_debugger*.

Here is an Example:

```
source path-to-kshdb/kshdb/dbg-trace.sh
# work, work, work.
# ... some ksh code

_Dbg_debugger
# start debugging here
```

Since *_Dbg_debugger* a function call, it can be nested inside some sort of conditional statement allowing one to be very precise about the conditions you want to debug under. And until first call to *_Dbg_debugger*, there is no debugger overhead.

Note that *_Dbg_debugger* causes the statement *after* the call to be stopped at.

CHAPTER 4

Command Syntax

4.1 Debugger Command Syntax

Command names and arguments are separated with spaces like POSIX shell syntax. Parenthesis around the arguments and commas between them are not used. If the first non-blank character of a line starts with #, the command is ignored.

Within a single command, tokens are then white-space split. Again, this process disregards quotes or symbols that have meaning in Python. Some commands like *eval*, have access to the untokenized string entered and make use of that rather than the tokenized list.

Resolving a command name involves possibly 3 steps. Some steps may be omitted depending on early success or some debugger settings:

1. The leading token is next looked up in the debugger alias table and the name may be substituted there. See "help alias" for how to define aliases, and "show alias" for the current list of aliases.

2. After the above, The leading token is looked up a table of debugger commands. If an exact match is found, the command name and arguments are dispatched to that command.

3. If after all of the above, we still don't find a command, the line may be evaluated as a ksh statement in the current context of the program at the point it is stoppped. However this is done only if "auto evaluation" is on. It is on by default.

If *auto eval* is not set on, or if running the Python statement produces an error, we display an error message that the entered string is "undefined".

If you want ksh shell command-processing, it's possible to go into an python shell with the corresponding the command *ksh* or *shell*. It is also possible to arrange going into an python shell every time you enter the debugger.

4.1.1 See also:

help syntax suffixes

4.2 Command suffixes which have special meaning

Some commands like *step*, or *list* do different things when an alias to the command ends in a particular suffix like >.

Here are a list of commands and the special suffixes:

command	suffix
list	>
step	+, -, <, >
next	+, -, <, >
quit	*!*
kill	*!*
eval	*?*

See the help on the specific commands listed above for the specific meaning of the suffix.

CHAPTER 5

Command Reference

Following *gdb*, we classify commands into categories. Note though that some commands, like *quit*, and *restart*, are in different categories and some categories are new, like *set*, *show*, and *info*.

5.1 Breakpoints

Making the program stop at certain points

A *breakpoint* can make your program stop at that point. You can set breakpoints with the break command and its variants. You can specify the place where your program should stop by file and line number or by function name.

The debugger assigns a number to each breakpoint when you create it; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use this number. Each breakpoint may be enabled or disabled; if disabled, it has no effect on your program until you enable it again.

The debugger allows you to set any number of breakpoints at the same place in your program. There is nothing unusual about this because different breakpoints can have different conditions associated with them.

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a condition for a breakpoint. A condition is just a Boolean expression in your programming language. A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is true.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated-that is, when the condition is false.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, pydb might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached.

Break conditions can be specified when a breakpoint is set, by adding a comma in the arguments to the break command. They can also be changed at any time with the *condition* command.

5.1.1 Break (set a breakpoint)

break [*loc-spec*]

Set a breakpoint at loc-spec.

If no location specification is given, use the current line.

Multiple breakpoints at one place are permitted, and useful if conditional.

Examples:

```
break               # Break where we are current stopped at
break 10            # Break on line 10 of the file we are
                    # currently stopped at
break /etc/profile:10   # Break on line 45 of os.path
```

See also:

tbreak, and *continue*.

5.1.2 Condition (add condition to breakpoint)

condition *bp_number condition*

bp_number is a breakpoint number. *condition* is a ksh expression which must evaluate to *True* before the breakpoint is honored. If *condition* is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

Examples:

```
condition 5 x > 10  # Breakpoint 5 now has condition x > 10
condition 5         # Remove above condition
```

See also:

break, *tbreak*.

5.1.3 Delete (remove breakpoints)

delete [*bpnumber* [*bpnumber*...]]

Delete some breakpoints.

Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. Without arguments, clear all breaks (but first ask confirmation).

See also:

clear

5.1.4 Disable (disable breakpoints)

disable *bpnumber* [*bpnumber* . . .]

Disables the breakpoints given as a space separated list of breakpoint numbers. See also *info break* to get a list.

See also:

enable

5.1.5 Enable (enable breakpoints)

enable *bpnumber* [*bpnumber* . . .]

Enables the breakpoints given as a space separated list of breakpoint numbers. See also *info break* to get a list.

See also:

disable, *tbreak*

5.1.6 Tbreak (temporary breakpoint)

tbreak [*location*] [**if** *condition*]

With a line number argument, set a break there in the current file. With a function name, set a break at first executable line of that function. Without argument, set a breakpoint at current location. If a second argument is *if*, subequent arguments given an expression which must evaluate to true before the breakpoint is honored.

The location line number may be prefixed with a filename or module name and a colon. Files is searched for using *sys.path*, and the *.py* suffix may be omitted in the file name.

Examples:

```
tbreak     # Break where we are current stopped at
tbreak 10  # Break on line 10 of the file we are currently stopped at
tbreak os.path.join # Break in function os.path.join
tbreak os.path:45   # Break on line 45 of os.path
tbreak myfile.py:45 # Break on line 45 of myfile.py
tbreak myfile:45    # Same as above.
```

See also:

break.

5.2 Data

Examining data.

5.2.1 Display (set display expression)

display [*format*] *expression*

Print value of expression *expression* each time the program stops. *format* may be used before *expression* and may be one of */c* for char, */x* for hex, */o* for octal, */f* for float or */s* for string.

For now, display expressions are only evaluated when in the same code as the frame that was in effect when the display expression was set. This is a departure from gdb and we may allow for more flexibility in the future to specify whether this should be the case or not.

With no argument, evaluate and display all currently requested auto-display expressions.

See also:

undisplay to cancel display requests previously made.

5.2.2 Eval (evaluate ksh expression)

eval *cmd*

eval

eval?

In the first form *cmd* is a string; *cmd* is a string sent to special shell builtin *eval*.

In the second form, use evaluate the current source line text.

Often when one is stopped at the line of the first part of an "if", "elif", "case", "return", "while" compound statement or an assignment statement, one wants to eval is just the expression portion. For this, use eval?. Actually, any alias that ends in ? which is aliased to eval will do thie same thing.

Run *cmd* in the context of the current frame.

If no string is given, we run the string from the current source code about to be run. If the command ends *?* (via an alias) and no string is given, the following translations occur:

```
{if|elif} <expr> [; then] => <expr>
while <expr> [; do]?      => <expr>
return <expr>             => <expr>
<var>=<expr>              => <expr>
```

The above is done via regular expression matching. No fancy parsing is done, say, to look to see if *expr* is split across a line or whether var an assignment might have multiple variables on the left-hand side.

Examples:

```
eval 1+2  # 3
eval      # Run current source-code line
eval?     # but strips off leading 'if', 'while', ..
          # from command
```

See also:

set autoeval and *examine*.

5.2.3 Examine

examine *expr1*

Print value of an expression via typeset, let, and failing these, eval.

Single variables and arithmetic expressions do not need leading `$` for their value is to be substituted. However if neither these, variables need `$` to have their value substituted.

In contrast to normal ksh expressions, expressions should not have blanks which would cause ksh to see them as different tokens.

Examples:

```
examine x+1   # ok
examine x + 1 # not ok
```

See also:

eval.

5.2.4 Load (source ksh file)

load *ksh-script*

Read in lines of a *ksh-script*.

See also:

info files.

5.2.5 Undisplay (cancel a display expression)

undisplay *display-number…*

Cancel some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to stop displaying.

No argument cancels all automatic-display expressions and is the same as *delete display*.

See also:

info display to see current list of display expresssions

5.3 Files

Specifying and examining files.

5.3.1 Edit

edit *position*

Edit specified file or module. With no argument, edits file containing most recent line listed.

See also:

list

5.3.2 List (show me the code!)

list[>**] [*location*|*.**|**-** [*num*]]

list *location* [*num*]

List source code.

Without arguments, print lines centered around the current line. If *num* is given that number of lines is shown.

If this is the first *list* command issued since the debugger command loop was entered, then the current line is the current frame. If a subsequent list command was issued with no intervening frame changing, then that is start the line after we last one previously shown.

A *location* is either:

- a number, e.g. 5,
- a filename, colon, and a number, e.g. */etc/profile:5*,
- a "." for the current line number
- a "-" for the lines before the current linenumber

If the location form is used with a subsequent parameter, the parameter is the starting line number is used. When there two numbers are given, the last number value is treated as a stopping line unless it is less than the start line, in which case it is taken to mean the number of lines to list instead.

Wherever a number is expected, it does not need to be a constant – just something that evaluates to a positive integer.

Examples:

```
list 5                  # List starting from line 5
list 4+1                # Same as above.
list /etc/profile:5     # List starting from line 5 of /etc/profile
list /etc/profile 5     # Same as above.
list /etc/profile 5 6   # list lines 5 and 6 of /etc/profile
list /etc/profile 5 2   # Same as above, since 2 < 5.
list profile:5 2        # List two lines starting from line 5 of profile
list .                  # List lines centered from where we currently are stopped
list -                  # List lines previous to those just shown
```

See also:

set listize, or *show listsize* to see or set the number of source-code lines to list.

5.4 Info

info [*info-subcommand*]

Get information on the program being debugged.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *info* for a list of info subcommands and what they do. Type `help info` for a summary list of info subcommands.

5.4.1 Info Breakpoints

info breakpoints [*bp-number...*]

Show status of user-settable breakpoints. If no breakpoint numbers are given, the show all breakpoints. Otherwise only those breakpoints listed are shown and the order given.

The "Num" column is the breakpoint number which can be used in an *enable*, *disable* or *condition* commads.

The "Disp" column contains one of "keep", "del", the disposition of the breakpoint after it gets hit.

The "enb" column indicates whether the breakpoint is enabled.

The "Where" column indicates where the breakpoint is located.

Example:

```
kshdb<4> info breakpoints
Num Type       Disp Enb What
1   breakpoint keep n   /etc/profile:8
2   breakpoint keep y   /etc/profile:10
    stop only if [[ ${PS1-} ]]
```

Show breakpoints.

See also:

break, *condition*, *delete*, *enable*, and ref:*disable <disable>*

5.4.2 Info Display

info display

Show all display expressions.

See also:

Display (set display expression)

5.4.3 Info Files

info files

Show a list of files that have been read in and properties regarding them.

See also:

Load (source ksh file)

5.4.4 Info Line

info line

Show information about the current line.

Example:

```
kshdb<1> info line
Line 4 of "/etc/profile"
```

See also:

info program, *info source*

5.4.5 Info Program

info program

Execution status of the program. Listed are:

- Reason the program is stopped.
- The next line to be run

Example:

```
kshdb<1> info program
Program stopped.
It stopped after being stepped.
Next statement to be run is:
[ "${PS1-}" ]
```

See also:

info line, and *info source*.

5.4.6 Info Source

info source

Information about the current ksh script file.

Example:

```
kshdb<1> info source
Current script file is /etc/profile
Located in /etc/profile
Contains 27 lines.
```

See also:

info program, *info files*, and *info line*.

5.4.7 Info Stack

info stack

An alias for **backtrace**

See also:

backtrace

5.4.8 Info Variables

info variables [*property*]

list global and static variable names.

Variable lists by property. *property* is an abbreviation of one of:

- arrays,
- exports,
- fixed,
- floats,
- functions,
- hash,
- integers, or
- readonly

Examples:

```
info variables              # show all variables
info variables readonly     # show only read-only variables
info variables integer      # show only integer variables
info variables functions    # show only functions
```

5.5 Running

Running, restarting, or stopping the program.

When a program is stopped there are several possibilities for further program execution. You can:

- terminate the program inside the debugger
- restart the program
- continue its execution until it would normally terminate or until a breakpoint is hit
- step exection which is runs for a limited amount of code before stopping

5.5.1 Continue (continue program execution)

continue [[*file* :] *lineno* | *function*]

Leave the debugger read-eval print loop and continue execution. Subsequent entry to the debugger however may occur via breakpoints or explicit calls, or exceptions.

If a line position or function is given, a temporary breakpoint is set at that position before continuing.

Examples:

```
continue           # Continue execution
continue 5         # Continue with a one-time breakpoint at line 5
continue basename # Go to os.path.basename if we have basename imported
continue /usr/lib/python2.7/posixpath.py:110 # Possibly the same as
                                             # the above using file
                                             # and line number
```

See also:

step jump, *next*, and finish provide other ways to progress execution.

5.5.2 Kill (send kill signal)

kill [*signal-number*]

Send this process a POSIX signal ('9' for 'SIGKILL' or `kill -SIGKILL`)

9 is a non-maskable interrupt that terminates the program. If program is threaded it may be expedient to use this command to terminate the program.

However other signals, such as 15 or `INT` that allow for the debugged to handle them can be sent.

Giving a negative number is the same as using its positive value.

Examples:

```
kill                # non-interuptable, nonmaskable kill
kill 9              # same as above
kill -9             # same as above
kill 15             # nicer, maskable TERM signal
kill -INT           # same as above
kill -SIGINT        # same as above
kill -WINCH         # send "window change" signal
kill -USR1          # send "user 1" signal
```

See also:

quit for less a forceful termination command, *run* restarts the debugged program.

5.5.3 Next (step over)

next [**+** | **-**] [*count*]

Step one statement ignoring steps into function calls at this level.

With an integer argument, perform *next* that many times. However if an exception occurs at this level, or we *return*, *yield* or the thread changes, we stop regardless of count.

A suffix of + on the command or an alias to the command forces to move to another line, while a suffix of - does the opposite and disables the requiring a move to a new line. If no suffix is given, the debugger setting 'different-line' determines this behavior.

See also:

skip, jump, *continue*, and finish provide other ways to progress execution.

5.5.4 Quit (gentle termination)

quit [*exit-code* [*shell-levels*]]

The program being debugged is aborted. If *exit-code* is given, then that will be the exit return code. If *shell-levels* is given, then up to that many nested shells are quit. However to be effective, the last of those shells should have been run under the debugger.

See also:

kill or *kill* for more forceful termination commands. *run* and restart are other ways to restart the debugged program.

5.5.5 Run (restart program execution)

run [*args*]

Attempt to restart the program.

See also:

quit, or *kill* for termination commands, or *set args* for another way to set run arguments.

5.5.6 Skip (skip over)

skip [*count*]

Skip (don't run) the next *count* command(s).

If *count* is given, stepping occurs that many times before stopping. Otherwise *count* is one. *count* can be an arithmetic expression.

Note that skipping doesn't change the value of $?. This has consequences in some compound statements that test on $?. For example in:

```
if grep foo bar.txt ; then
   echo not skipped
fi
```

Skipping the *if* statement will, in effect, skip running the *grep* command. Since the return code is 0 when skipped, the *if* body is entered. Similarly the same thing can happen in a *while* statement test.

See also:

next, *step*, and *continue* provide other ways to progress execution.

5.5.7 Step (step into)

step [**+** | **-** [*count*]

Execute the current line, stopping at the next event.

With an integer argument, step that many times.

You can Set an event, by suffixing one of the symbols +, -, or after the command or on an alias of that. A suffix of + on a command or an alias forces a move to another line, while a suffix of - disables this requirement.

If no suffix is given, the debugger setting *different-line* determines this behavior.

Examples:

```
step        # step 1 event, *any* event
step 1      # same as above
step 5/5+0  # same as above
step+
step-
```

See also:

next command. *skip*, and *continue* provide other ways to progress execution.

set [*set-subcommand*]

Modifies parts of the debugger environment.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *set* for a list of set subcommands and what they do. Type `help set` for a summary list of set subcommands.

All of the "set" commands have a corresponding *show* command.

5.6 Set

Modifies parts of the debugger environment. You can see these environment settings with the *show* command.

5.6.1 Set Annotate (GNU Emacs annotation level)

set annotate { **0** | **1** }

Set annotation level. This is a (mostly obsolete) gdb setting, but it is used in GNU Emacs.

```
0 - normal
1 - fullname (for use when running under GNU Emacs).
```

See also:

show annotate

5.6.2 Set Args (progam arguments)

set args [*script-args*]

Set argument list to give program being debugged when it is started. Follow this command with any number of args, to be passed to the program.

See also:

run

5.6.3 Set Auto Eval (auto evaluation of unrecognized debugger commands)

set autoeval [**on** | **off**]

Evaluate unrecognized debugger commands.

Often inside the debugger, one would like to be able to run arbitrary ksh commands without having to preface expressions with `print` or `eval`. Setting *autoeval* on will cause unrecognized debugger commands to be *eval*'d as a ksh expression.

Note that if this is set, on error the message shown on type a bad debugger command changes from:

```
Undefined command: "fdafds". Try "help".
```

to something more ksh-eval-specific such as:

```
/tmp/kshdb_eval_26397:2: command not found: fdafds
```

See also:

show autoeval

5.6.4 Set Auto List

set autolist [on | off]

Run the *list* command every time you stop in the debugger.

With this, you will get output like:

```
-> 1 from subprocess import Popen, PIPE
(trepan2) next
(/users/fbicknel/Projects/disk_setup/sqlplus.py:2): <module>
** 2 import os
  1     from subprocess import Popen, PIPE
  2  -> import os
  3     import re
  4
  5     class SqlPlusExecutor(object):
  6         def __init__(self, connection_string='/ as sysdba', sid=None):
  7             self.__connection_string = connection_string
  8             self.session = None
  9             self.stdout = None
 10             self.stderr = None
(trepan2) next
(/users/fbicknel/Projects/disk_setup/sqlplus.py:3): <module>
** 3 import re
  1     from subprocess import Popen, PIPE
  2     import os
  3  -> import re
  4
  5     class SqlPlusExecutor(object):
  6         def __init__(self, connection_string='/ as sysdba', sid=None):
  7             self.__connection_string = connection_string
  8             self.session = None
  9             self.stdout = None
 10             self.stderr = None
(trepan2)
```

You may also want to put this this in your debugger startup file. See Startup Profile

See also:

show autolist

5.6.5 Set Basename (basename only in file path)

set basename [**on** | **off**]

Set short filenames in debugger output.

Setting this causes the debugger output to give just the basename for filenames. This is useful in debugger testing or possibly showing examples where you don't want to hide specific filesystem and installation information.

See also:

show basename

5.6.6 Set Cmdtrace (show debugger commands before running)

set cmdtrace [**on** | **off**]

Set echoing lines read from debugger command files

See also:

show cmdtrace

5.6.7 Set Confirm (confirmation of potentially dangerous operations)

set confirm [**on** | **off**]

Set confirmation of potentially dangerous operations.

Some operations are a bit disruptive like terminating the program. To guard against running this accidentally, by default we ask for confirmation. Commands can also be exempted from confirmation by suffixing them with an exclamation mark (!).

See also:

show confirm

5.6.8 Set Different (consecutive stops on different file/line positions)

set different [**on** | **off**]

Set consecutive stops must be on different file/line positions. If no argument is given, different is set "off".

One of the challenges of debugging is getting the granualarity of stepping comfortable. By setting different "on" you can set a more coarse-level of stepping which often still is small enough that you won't miss anything important.

Note that the *step* and *next* debugger commands have '+' and '-' suffixes if you wan to override this setting on a per-command basis.

See also:

set trace to change what events you want to filter. show trace.

5.6.9 Set Editing (readline editing of commands)

set editing [**on** | **off** | **emacs** | **gmacs** | **vi**]

Readline editing of command lines.

See also:

show editing

5.6.10 Set Highlight (terminal highlighting)

set highlight [**reset**] {**plain** | **light** | **dark** | **off**}

Set whether we use terminal highlighting for ANSI 8-color terminals. Permissible values are:

plain no terminal highlighting

off same as plain

light terminal background is light (the default)

dark terminal background is dark

If the first argument is *reset*, we clear any existing color formatting and recolor all source code output.

A related setting is *style* which sets the Pygments style for terminal that support, 256 colors. But even here, it is useful to set the highlight to tell the debugger for bold and emphasized text what values to use.

Examples:

```
set highlight off   # no highlight
set highlight plain # same as above
set highlight       # same as above
set highlight dark  # terminal has dark background
set highlight light # terminal has light background
set highlight reset light # clear source-code cache and
                          # set for light background
set highlight reset # clear source-code cache
```

See also:

show highlight and *set style*

5.6.11 Set Listsize (list command line count)

set listsize *number-of-lines*

Set the number lines printed in a *list* command by default

See also:

show listsize

5.6.12 Set Style (pygments formatting style)

set style [*pygments-style*]

Set the pygments style in to use in formatting text for a 256-color terminal. Note: if your terminal doesn't support 256 colors, you may be better off using *–highlight=plain* or *–highlight=dark* instead. To turn off styles use *set style none*.

To list the available pygments styles inside the debugger, omit the style name.

Examples:

```
set style monokai # use monokai style (a dark style)
set style         # list all known pygments styles
set style off     # turn off any pygments source mark up
```

See also:

show style and *set highlight*

5.6.13 Set Width (terminal width)

set width *number*

Set the number of characters the debugger thinks are in a line.

See also:

show width

5.7 Stack

Examining the call stack.

The call stack is made up of stack frames. The debugger assigns numbers to stack frames counting from zero for the innermost (currently executing) frame.

At any time the debugger identifies one frame as the "selected" frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops, the debugger selects the innermost frame. The commands below can be used to select other frames by number or address.

5.7.1 Backtrace (show call-stack)

backtrace [*count*]

Print a stack trace, with the most recent frame at the top. With a positive number, print at most many entries. With a negative number print the top entries minus that number.

An arrow indicates the 'current frame'. The current frame determines the context used for many debugger commands such as expression evaluation or source-line listing.

Examples:

```
backtrace    # Print a full stack trace
backtrace 2  # Print only the top two entries
backtrace -1 # Print a stack trace except the initial (least recent) call.
```

5.7.2 Frame (absolute frame positioning)

frame [*thread-Name*|*thread-number*] [*frame-number*]

Change the current frame to frame *frame-number* if specified, or the current frame, 0, if no frame number specified.

If a thread name or thread number is given, change the current frame to a frame in that thread. Dot (.) can be used to indicate the name of the current frame the debugger is stopped in.

A negative number indicates the position from the other or least-recently-entered end. So *frame -1* moves to the oldest frame, and *frame 0* moves to the newest frame. Any variable or expression that evaluates to a number can be used as a position, however due to parsing limitations, the position expression has to be seen as a single blank-delimited parameter. That is, the expression *(5*3)-1* is okay while *(5 * 3) - 1)* isn't.

Examples:

```
frame     # Set current frame at the current stopping point
frame 0   # Same as above
frame 5-5 # Same as above. Note: no spaces allowed in expression 5-5
frame .   # Same as above. "current thread" is explicit.
frame . 0 # Same as above.
frame 1   # Move to frame 1. Same as: frame 0; up
frame -1  # The least-recent frame
frame MainThread 0 # Switch to frame 0 of thread MainThread
frame MainThread   # Same as above
frame -2434343 0   # Use a thread number instead of name
```

See also:

down, *up*, *backtrace*, and info threads.

5.7.3 Up (relative frame motion towards a less-recent frame)

up [*count*]

Move the current frame up in the stack trace (to an older frame). 0 is the most recent frame. If no count is given, move up 1.

See also:

down and *frame*.

5.7.4 Down (relative frame motion towards a more-recent frame)

down [*count*]

Move the current frame down in the stack trace (to a newer frame). 0 is the most recent frame. If no count is given, move down 1.

See also:

up and *frame*.

show [*subcommand*]

A command for showing things about the debugger. You can give unique prefix of the name of a subcommand to get information about just that subcommand. nn Type *show* for a list of show subcommands and what they do. Type `help show` for a summmary list of show subcommands. Many of the "show" commands have a corresponding *set* command.

5.8 Show

5.8.1 Show Aliases (debugger command aliases)

show aliases [*alias* . . . | *]

Show command aliases. If parameters are given a list of all aliases and the command they run are printed. Alternatively one can list specific alias names for the commands those specific aliases are attached to. If instead of an alias "*" appears anywhere as an alias then just a list of aliases is printed, not what commands they are attached to.

See also:

alias

5.8.2 Show Annotate (GNU Emacs annotation level)

show annotate

Show annotation level. This is a (mostly obsolete) gdb setting, but it is used in GNU Emacs.

```
0 - normal
1 - fullname (for use when running under GNU Emacs).
```

See also:

set annotate

5.8.3 Show Args (arguments when program is started)

show args

Show the argument list to give debugged program when it is started

5.8.4 Show Autoeval (auto evaluation of unrecognized debugger commands)

show autoeval

Show whether ksh evaluate of unrecognized debugger commands.

See also:

set autoeval

5.8.5 Show Autolist

show autolist

Run a debugger ref:*list* <*list*> command automatically on debugger entry.

See also:

set autolist

5.8.6 Show Basename (basename only in file path)

show basename

Show whether filename basenames or full path names are shown.

See also:

set basename

5.8.7 Show Cmdtrace (show debugger commands before running)

show cmdtrace

Show debugger commands before running them

See also:

set cmdtrace

5.8.8 Show Confirm (confirmation of potentially dangerous operations)

show confirm

Show confirmation of potentially dangerous operations

See also:

show confirm

5.8.9 Show Different (consecutive stops on different file/line positions)

Show consecutive stops on different file/line positions

See also:

set different

5.8.10 Show Editing (readline editing)

show editing

Show editing of command lines as they are typed.

See also:

set editing

5.8.11 Show Highlight (terminal highlighting)

show highlight

Show whether we use terminal highlighting.

See also:

set highlight

5.8.12 Show Listsize (list command line count)

show listsize

Show the number lines printed in a *list* command by default

See also:

set listsize

5.8.13 Show Style (pygments formatting style)

show style *pygments-style*

Show the pygments style used in formatting 256-color terminal text.

See also:

set style and *show highlight*

5.8.14 Show Width (terminal width)

show width

Show the number of characters the debugger thinks are in a line.

See also:

set width

5.9 Support

5.9.1 Alias (add debugger command alias)

alias *alias-name debugger-command*

Add alias *alias-name* for a debugger command *debugger-comand*.

Add an alias when you want to use a command abbreviation for a command that would otherwise be ambigous. For example, by default we make `s` be an alias of `step` to force it to be used. Without the alias, `s` might be `step`, `show`, or `set` among others

Example:

```
alias cat list   # "cat myprog.py" is the same as "list myprog.py"
alias s   step   # "s" is now an alias for "step".
                 # The above example is done by default.
```

See also:

unalias and *show alias*.

5.9.2 Help (Won't you please help me if you can)

help [*command* [*subcommand*]|*expression*]

Without argument, print the list of available debugger commands.

When an argument is given, it is first checked to see if it is command name.

Some commands like *info*, *set*, and *show* can accept an additional subcommand to give help just about that particular subcommand. For example *help info line* give help about the info line command.

See also:

examine and whatis.

5.9.3 Source (Read and run debugger commands from a file)

source [**-v**][**-Y****|-N**][**-c**] *file*

Read debugger commands from a file named *file*. Optional *-v* switch (before the filename) causes each command in *file* to be echoed as it is executed. Option *-Y* sets the default value in any confirmation command to be "yes" and *-N* sets the default value to "no".

Note that the command startup file *.trepanc* is read automatically via a *source* command the debugger is started.

An error in any command terminates execution of the command file unless option *-c* is given.

5.9.4 Unalias (remove debugger command alias)

unalias *alias-name*

Remove alias *alias-name*.

See also:

alias.

Contents

CHAPTER 6

kshdb command

6.1 Synopsis

kshdb [*debugger-options*] [–] [*ksh-script* [*script-options* . . .]]

kshdb [*options*] -c *execution-string*

6.2 Description

kshdb is a *ksh* script to which arranges for another *ksh* script to be debugged.

The debugger has a similar command interface as gdb.

If your ksh script needs to be passed options, add −− before the script name. That will tell *kshdb* not to try to process any further options.

6.3 Options

-h | –help

Print a usage message on standard error and exit with a return code of 100.

-A | –annotation *level*

Sets to output additional stack and status information which allows front-ends such as Emacs to track what's going on without polling.

This is needed in for regression testing. Using this option is equivalent to issuing:

```
set annotate LEVEL
```

inside the debugger. See *set annotate* for more information on that command

-B | –basename

In places where a filename appears in debugger output give just the basename only. This is needed in for regression testing. Using this option is equivalent to issuing:

```
set basename on
```

inside the debugger. See *set basename* for more information on that command

-n | –nx | –no-init

Normally the debugger will read debugger commands in *~/.kshdbinit* if that file exists before accepting user interaction. *.kshdbinit* is analogous to GNU gdb's *.gdbinit*: a user might want to create such a debugger profile to add various user-specific customizations.

Using the *-n* option this initialization file will not be read. This is useful in regression testing or in tracking down a problem with one's *.kshdbinit* profile.

-c | –command *command-string*

Instead of specifying the name of a script file, one can give an execution string that is to be debugged. Use this option to do that.

-q | –quiet

Do not print introductory version and copyright information. This is again useful in regression testing where we don't want to include a changeable copyright date in the regression-test matching.

-x | –eval-command *debugger-cmdfile*

Run the debugger commands *debugger-cmdfile* before accepting user input. These commands are read however after any *.kshdbinit* commands. Again this is useful running regression-testing debug scripts.

-L | –library *debugger-library*

The debugger needs to source or include a number of functions and these reside in a library. If this option is not given the default location of library is relative to the installed kshdb script: *../lib/kshdb*.

-T | –tempdir *temporary-file-directory*

The debugger needs to make use of some temporary filesystem storage to save persistent information across a subshell return or in order to evaluate an expression. The default directory is */tmp* but you can use this option to set the directory where debugger temporary files will be created.

-t | –tty *tty-name*

Debugger output usually goes to a terminal rather than stdout or stdin which the debugged program may use. Determination of the tty or pseudo-tty is normally done automatically. However if you want to control where the debugger output goes, use this option.

-V | –version

Show version number and no-warranty and exit with return code 1.

Bugs <—-

The way this script arranges debugging to occur is by including (or actually "source"-ing) some debug-support code and then sourcing the given script or command string.

One problem with sourcing a debugged script is that the program name stored in `$0` will not be the name of the script to be debugged. The debugged script will appear in a call stack not as the top item but as the item below *kshdb*.

The *kshdb* script option assumes a version of _ksh_ with debugging support, vesion 2014-12-24 or later.

The debugger slows things down a little because the debugger has to intercept every statement and check to see if some action is to be taken.

6.4 See also

- kshdb github - the github project page
- zshdb - a similar POSIX shell debugger for zsh
- bashdb - a similar POSIX shell debugger for bash

6.5 Author

The current version is maintained (or not) by Rocky Bernstein.

6.6 Copyright

CHAPTER 7

Indices and tables

- genindex
- search

Index

T

U