Chapter 9. Debugging Shell Programs

By Hansika Bhambhaney

OBJECTIVES:

1.Understand how to **trace and debug bash scripts** using built-in `set` options like `-x` (execution trace) and `-v` (input printing).

2.Learn to **simulate signal handling** and introduce manual trace points.

3.Implement and use a **custom bash debugger** that can simulate stepping and breakpoints.

4.Gain practical experience through **examples and exercises** in interactive debugging of shell scripts.

9.1 `set` Options for Tracing Execution

Bash provides three key options to aid debugging:

1. **`-n` or `noexec`**:
    - Checks for **syntax errors** without executing any commands.
    - Useful for complex scripts where execution might have side effects.
    - Note: Once `-n` is set, it cannot be unset during the script's execution.
2. **`-v` or `verbose`**:
    - Echoes commands **before** executing them.
    - Helpful to locate where a script fails by showing input lines.
3. **`-x` or `xtrace`**:
    - Echoes commands **after** expansions, showing actual executed commands.
    - Outputs each command with a `+` sign (customizable using `PS4`).

Example:

```
set -x
var="Hello"
echo "$var"
set +x
```

We can use `set -o option` to enable and `set +o option` to disable each option as needed.

set -o option	**Command-line option**	**Action**

The verbose option simply echoes (to standard error) whatever input the shell gets. It is useful for finding the exact point at which a script is bombing

`set -x` **(xtrace)**

- **Purpose**: Displays **each command as executed**, with variable substitution shown.

- **Use Case**: Best for **tracking the exact value** of variables and command evaluation.

- **Example**:

OUTPUT

```
set -x
x=5
y=10
z=$((x + y))
echo "Sum: $z"
set +x
```

```
+ x=5
+ y=10
+ z=15
+ echo 'Sum: 15'
Sum: 15
```

Customize Output Using PS4
We can define how debug output appears by setting the PS4 variable.

Example:

```
export PS4='+($LINENO): '
set -x
echo "Debugging line"
set +x
```

OUTPUT:

```
+(1): echo "Debugging line"
Debugging line
```

9.2 Fake Signals for Debugging with `trap`

trap allows you to run a command or function when a signal occurs, including "fake signals" which Bash uses internally for debugging.

SIGNAL	TRIGGER
EXIT	When the script exits (normal or error)
ERR	After a command fails (non-zero exit)
DEBUG	Before every command is executed
RETURN	When a function or sourced file returns

EXIT
The EXIT trap, when set, will run its code whenever the script within which it was set exits.

Example:
trap 'echo exiting from the script' EXIT

```
 echo 'start of the script'
```

Output:

```
start of the script
exiting from the script
```

An EXIT trap occurs no matter how the script exits—whether normally (by finishing the last statement), by an explicit exit or return statement, or by receiving a "real" signal such as INT or TERM.

PROGRAM:

```
#!/bin/bash

# Set a trap to print a thank-you message when the script exits
trap 'echo "Thank you for playing!"' EXIT

# Generate a random number between 1 and 10
magicnum=$((RANDOM % 10 + 1))

echo "Guess a number between 1 and 10:"

# Start reading guesses
while true; do
   read -p "Guess: " guess

   # Add a delay for suspense
   sleep 2

   if [ "$guess" -eq "$magicnum" ] 2>/dev/null; then
      echo "Right!"
      exit
   else
      echo "Wrong! Try again."
   fi
done
```

This program picks a number between 1 and 10 by getting a random number (the built-in variable RANDOM), extracting the last digit (the remainder when divided by 10), and adding 1. Then it prompts you for a guess, and after 4 seconds, it will tell you if you guessed right.

ERR

The fake signal ERR enables you to run code whenever a command in the surrounding script or function exits with non-zero status. Trap code for ERR can take advantage of the built-in variable ?, which holds the exit status of the previous command. It survives the trap and is accessible at the beginning of the trap-handling code. A simple but effective use of this is to put the following code into a script you want to debug:

```
function errtrap {
es=$?
 echo "ERROR: Command exited with status $es."
}
 trap errtrap ERR
```

The first line saves the nonzero exit status in the local variable es.

EXAMPLE CODE:

```
function errtrap {
 es=$?
echo "ERROR: Command exited with status $es."
 }
trap errtrap ERR
 function bad {
 return 17
}
 Bad
```

This only prints ERROR: Command exited with status 17

This simple code is actually not a bad all-purpose debugging mechanism. It takes into account that a nonzero exit status does not necessarily indicate an undesirable condition or event: remember that every control construct with a conditional (if, while, etc.) uses a nonzero exit status to mean "false." Accordingly, the shell doesn't generate ERR traps when statements or expressions in the "condition" parts of control structures produce nonzero exit statuses. Also, an ERR trap is not inherited by shell functions, command substitutions, and commands executed in a subshell. However this inheritance behaviour can be turned on by using set -o errtrace

DEBUG

Another fake signal, DEBUG, causes the trap code to be executed before every statement in a function or script.

SYNTAX:

```
function dbgtrap
 {
echo "badvar
is
$badvar"
}
trap dbgtrap DEBUG ...section of code in which the problem occurs...
trap - DEBUG # turn off the DEBUG trap
```

This code will print the value of the wayward variable before every statement between the two traps.

DEBUG trap and then calls a function, the statements within the function will not execute the trap. There are three ways around this. Firstly we can set a trap for DEBUG explicitly within the function. Alternately we can declare the function with the -t option which turns on debug inheritance in functions and allows a function to inherit a DEBUG trap from the caller.

RETURN

A RETURN trap is executed each time a shell function or a script executed with the . or source commands finishes executing.

Example of a RETURN trap

```
function returntrap
{
echo "A return occurred"
}
trap returntrap RETURN
 function hello {
 echo "hello world"
 } hello
```

1.When the script is executed it executes the hello function

2.Used for debugging **function return points**.

```
myfunc() {
    echo "Inside function"
}
trap 'echo "Function returned at line $LINENO"' RETURN
myfunc
```

In addition to these fake signals, bash 3.0 added some other features to help with writing a full-scale debugger for bash. The first of these is the extdebug option to the shopt command, which switches on certain things that are useful for a debugger. These include:

• The -F option to declare displays the source filename and line number corresponding to each function name supplied as an argument.

• If the command that is run by the DEBUG trap returns a non-zero value, the next command is skipped and not executed.

• If the command run by the DEBUG trap returns a value of 2, and the shell is executing in a subroutine (a shell function or a shell script executed by the . or source commands), a call to return is simulated. The shell also has a new option, —debugger, which switches on both the extdebug and functrace functionality.

9.3. Custom Variables for Tracing

These special Bash variables are critical for writing a custom debugger or tracing tool.

- `BASH_SOURCE`

- Array of filenames corresponding to the current function call stack.
- `BASH_SOURCE[0]` is the current script/function.

🔸 `BASH_LINENO`

- Array of line numbers where functions were called.
- Helps track which line a function was invoked from.

🔸 `FUNCNAME`

- Array of currently executing function names.
- `FUNCNAME[0]` is the current function.

🔸 `BASH_ARGC`, `BASH_ARGV`

- `BASH_ARGC[i]` gives the argument count of function at call level `i`.
- `BASH_ARGV` gives arguments in reverse call order.

Example: Function Stack Debug Info

```
debug() {
  echo "Function: ${FUNCNAME[1]} called from line ${BASH_LINENO[0]} in ${BASH_SOURCE[1]}"
}

myfunc() {
  debug
  echo "Inside myfunc"
}

myfunc
```

OUTPUT:

```
Function: myfunc called from line 2 in ./myscript.sh
Inside myfunc
```

Structure and Design of a Simple Bash Debugger

The `bashdb` debugger is a minimalist shell script debugger that:

- Steps through code, one line at a time.
- Sets breakpoints at specific lines.
- Inspects and changes variables during script execution.
- Displays source code with indicators for breakpoints and current line.
- Works without modifying the original script.

It works by:

1. Combining (concatenating) the target script (guinea pig) with a debugging preamble.
2. Running this new temporary file, so the original script remains untouched.

It has 3 parts:

1. Driver Script (`bashdb`) – sets up and starts debugging.
2. **Preamble (`bashdb.pre`)** – inserted at the beginning of the target script.
3. **Debugger Functions (`bashdb.fns`)** – contain actual logic for stepping, breakpoints, tracing, etc.

Driver script:

The driver script is called `bashdb`.
 Its job is to:
1️⃣Check arguments and file validity.
2️⃣Concatenate the **preamble** and the **target script** into a temporary file.
3️⃣Run this new combined script in place of itself.

SCRIPT:

```
# bashdb - a bash debugger

# Driver Script: concatenates the preamble and the target

# and then executes the new script.



echo 'bash Debugger version 1.0'



_dbname=${0##*/}



if (( $# < 1 )) ; then

    echo "$_dbname: Usage: $_dbname filename" >&2

    exit 1
```

```
fi


_guineapig=$1


if [ ! -r $1 ]; then

    echo "$_dbname: Cannot read file '$_guineapig'." >&2

    exit 1

fi


shift


_tmpdir=/tmp

_libdir=.

_debugfile=$_tmpdir/bashdb.$$


cat $_libdir/bashdb.pre $_guineapig > $_debugfile


exec bash $_debugfile $_guineapig $_tmpdir $_libdir "$@"
```

`echo 'bash Debugger version 1.0'`

- **Purpose**: Prints a header so the user knows they're using the debugger.
- Not functionally necessary, purely informational.

`_dbname=${0##*/}`

- **Purpose**: Extracts just the script name from `$0` (which may include a path).
- `${0##*/}` means:
  - Take the value of `$0`,
  - Remove everything up to and including the last `/`,
  - Leaving just the basename.
- **Use case**: Used later in error messages for user-friendly display.

`if (( $# < 1 )) ; then ... fi`

- Checks that at least **one argument** is provided (the script to debug).
- `$#` is the count of arguments given to the script.

`_guineapig=$1`

- Assigns the **first argument** to `_guineapig`.
- This is the filename of the script that we want to debug.

`if [ ! -r $1 ]; then ... fi`

- Checks if the **target script is readable** (`-r` test).

`shift`

- Shifts all arguments left by one:
  - `$2` becomes `$1`, `$3` becomes `$2`, etc.
- After shifting:
  - User's **positional arguments to the target script** are now in `$1`, `$2`, ...
- **Why?** When we pass arguments to the target script later, they should see only their arguments.

`_tmpdir=/tmp`

- Temporary directory for storing the combined script file.
- `/tmp` is standard on Unix/Linux systems for temporary files.

`_libdir=.`

- Directory where `bashdb.pre` and other debugger helper files live.
- Here, set to current directory (`.`).

`_debugfile=$_tmpdir/bashdb.$$`

- Name for the temporary combined script file.
- `$$` = Process ID of the running shell — guarantees **unique filename**, so multiple debugger sessions don't collide.
- Example: `/tmp/bashdb.4567`.

```
cat $_libdir/bashdb.pre $_guineapig > $_debugfile
```

- **Core step**!
- Combines the **debug preamble** file and the **target script** into one temporary file:
    1. `$_libdir/bashdb.pre`: Debug logic, traps, initializations.
    2. `$_guineapig`: The user's actual script.
- Output (`>`) into `_debugfile`.

9.4 PREAMBLE

The preamble is a block of code that gets automatically added to the beginning of your target script (called the "guinea pig"). It serves as the initialization and setup for debugging.

This preamble is stored in the file `bashdb.pre`, and is essential for activating debugging features like:

- Breakpoint support
- Step-by-step execution
- Clean-up after execution
- Debugging line numbering
- Tracing script lines and showing them with context

CODE:

```
# bashdb preamble

# This file gets prepended to the shell script being debugged.

# Arguments:

# $1 = the name of the original guinea pig script

# $2 = the directory where temporary files are stored

# $3 = the directory where bashdb.pre and bashdb.fns are stored

_debugfile=$0

_guineapig=$1

_tmpdir=$2

_libdir=$3

shift 3

source $_libdir/bashdb.fns

_linebp=

let _trace=0

let _i=1

while read; do

    _lines[$_i]=$REPLY

    let _i=$_i+1

done < $_guineapig


trap _cleanup EXIT

let _steps=1
```

```
trap '_steptrap $(( $LINENO - 29 ))' DEBUG
```

Comments

The comments at the top simply describe:

- This file is a **preamble** that gets added to the beginning of the script being debugged.
- The arguments it receives:
  - `$1`: Original script name
  - `$2`: Temporary directory path
  - `$3`: Directory containing debugging support files

Assigning arguments to variables

The preamble starts by assigning the first few arguments to named variables for clarity:

- `_debugfile`: The temporary combined script file (actually `$0`, the currently running script).
- `_guineapig`: The original script you want to debug.
- `_tmpdir`: Directory for temporary files (usually `/tmp`).
- `_libdir`: Where debugging functions (`bashdb.fns`) are stored.

After this, it runs `shift 3` to remove these initial arguments from the positional parameters so any further arguments can be used as inputs to the script itself (e.g., command-line parameters).

Loading debugger functions

It then **sources** (`source`) the `bashdb.fns` file, which contains all the actual debugger logic.
These functions handle:

- Stepping
- Breakpoints
- Cleanup
- Command loop interactions

This keeps the preamble short and modular.

Initializing internal variables

The preamble sets up a few variables:

- `_linebp`: Stores the line number of any set breakpoint.
- `_trace`: Controls whether to display each command as it is executed (off by default).
- `_i`: A counter for reading lines.

Reading the target script into an array

The preamble **reads each line** of the original script (`_guineapig`) and stores them in an array (`_lines[]`), indexed by `_i`.

- This allows the debugger to later **display or list** source code lines during interactive debugging sessions.

- It uses `$REPLY` to preserve exact whitespace.

Setting up cleanup

A trap is installed on `EXIT` so that when the script finishes (whether it exits normally or due to an error), a `_cleanup` function (from `bashdb.fns`) is called.
This typically deletes the temporary combined script and does any necessary final cleanup.

Preparing stepping

The variable `_steps` is set to 1.
This tells the debugger to **stop after executing the first line**, so that the user can start stepping through interactively right from the beginning.

Setting up the DEBUG trap

A `DEBUG` trap is set so that before **every command**, a function `_steptrap` is executed.

- The argument to `_steptrap` is the **line number**, adjusted by subtracting the number of lines in the preamble (here ~29 lines) to match the line numbers in the original script.

- The `_steptrap` function checks whether to stop (based on breakpoints or step count), or to allow execution to continue.

9.5 Debugger functions file (`bashdb.fns`)

What is `bashdb.fns`?

It is a **separate file** containing all the **core functions** that actually implement the debugger logic. It is sourced in the preamble:

```
source $_libdir/bashdb.fns
```

The preamble is like the **setup script**:

- It initializes variables.
- Loads the functions.
- Reads the script into an array.
- Sets traps.

But it **does not define the functions themselves** — those are in `bashdb.fns`.

What does `bashdb.fns` contain?

It includes key functions such as:

🔹 **`_steptrap`**

Runs before every line (thanks to `DEBUG` trap). Checks:

- Are we stepping?
- Did we hit a breakpoint?

🔹 **`_cmdloop`**

Interactive loop where you type commands like:

- `s` → step
- `g` → go (continue)
- `bp <line>` → set breakpoint
- `cb` → clear breakpoint
- `ds` → display script
- `!` → run a shell command

🔹 **`_cleanup`**

Removes temporary files and resets states when the script exits.

🔹 **`_setbp`**

Handles setting breakpoints.

🔹 **`_setbc`**

Handles break conditions (optional).

🔹 **Other helper functions**

E.g., displaying code lines, printing help messages, etc.

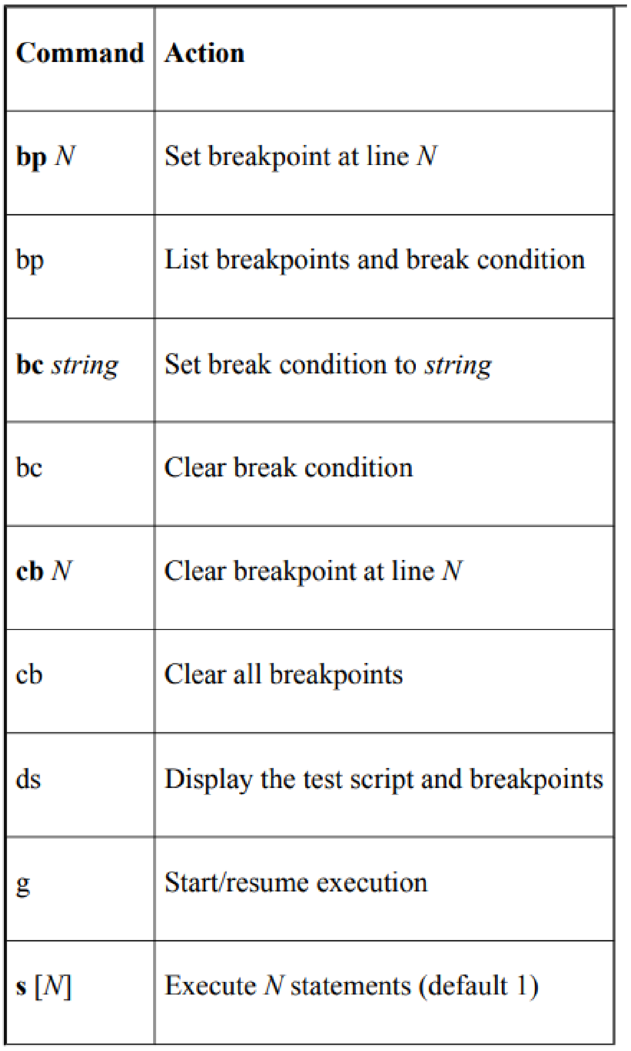
After the functions file?

Once the preamble and functions are set up:

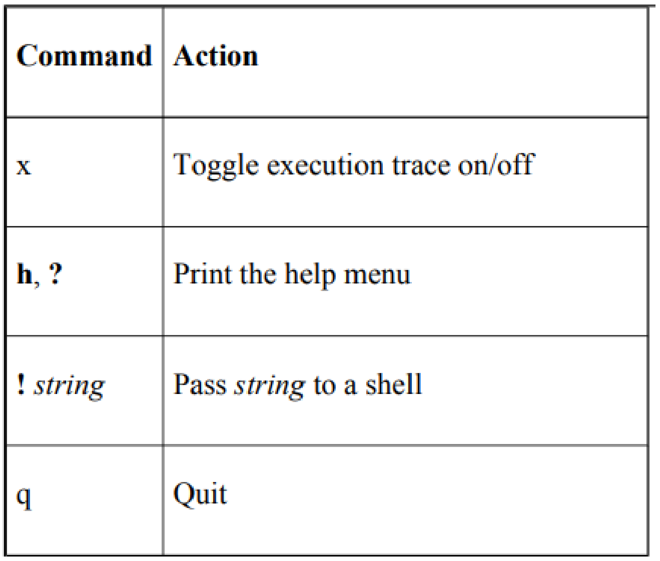
1. The debugger environment is ready.
2. Your original script lines (the "guinea pig") start executing, line by line,

under debugger control.
3. You control it interactively via commands from `_cmdloop`.

Bashdb commands



Command	**Action**
bp *N*	Set breakpoint at line *N*
bp	List breakpoints and break condition
bc *string*	Set break condition to *string*
bc	Clear break condition
cb *N*	Clear breakpoint at line *N*
cb	Clear all breakpoints
ds	Display the test script and breakpoints
g	Start/resume execution
s [*N*]	Execute *N* statements (default 1)



Command	**Action**
x	Toggle execution trace on/off
h, **?**	Print the help menu
! *string*	Pass *string* to a shell
q	Quit

steptrap runs after every statement in the guinea pig as a result of the trap on DEBUG in the preamble. If a breakpoint has been reached or the user previously typed in a step command(s), _steptrap calls the command loop. In doing so, it effectively "interrupts" the shell that is running the guinea pig to hand control over to the user.

Stepping

Step meaning

When you type `s` in the debugger command loop, it means "step" — run a certain number of single statements, one at a time.

- If you omit an argument (just `s`), it defaults to 1 step.
- If you provide an argument, like `s 3`, it means "step 3 lines/statements".

How `_steps` is used

- `_steps` is a variable that controls how many steps (lines) to run before stopping again.
- It is set by the `s` command in the command loop.

Case 1: User types `s` with no argument (default to 1)

Flow:

1. The command loop sets `_steps=1`.
2. The command loop exits, returning to `_steptrap`.
3. `_steptrap` also exits, giving control back to the shell.
4. The shell executes the next line of the script.
5. After that line, the `DEBUG` trap is triggered again → `_steptrap` is called.
6. In `_steptrap`, `_steps` is decremented from 1 to 0.
7. Now `_steps` is 0:

- A "stopped" message is printed.
  The command loop is invoked again, waiting for your next

Case 2: User types `s 3` (step 3 lines)

Flow:

1. The command loop sets `_steps=3`.
2. Command loop exits → back to `_steptrap`.
3. `_steptrap` exits → back to shell → one line runs.
4. After running, `DEBUG` trap triggers `_steptrap`.

- `_steps` is decremented from 3 to 2.
- `_steptrap` exits again → run next line.

   5. Repeat:

- Next line runs, `_steps` becomes 1.

   6 Repeat:

- Third line runs, `_steps` becomes 0.

7. Now, since `_steps` is 0:

- The "stopped" message is printed.
- The command loop is started again, waiting for your input

BREAKPOINTS

The bp command calls the function _setbp, which can do two things, depending on whether an argument is supplied or not. Here is the code for _setbp**:**

```
# Set a breakpoint at the given line number or list breakpoints

function _setbp {

    local i

    if [ -z "$1" ]; then

        _listbp
```

```
  elif [[ "$1" =~ ^[0-9]+$ ]]; then
    if [ -n "${_lines[$1]}" ]; then
      # Add new line number to _linebp array, avoiding duplicates
      _linebp=($(echo "${_linebp[*]} $1" | tr ' ' '\n' | sort -n | uniq))
      _msg "Breakpoint set at line $1"
    else
      _msg "Breakpoints can only be set on non-blank lines"
    fi
  else
    _msg "Please specify a numeric line number"
  fi
}
```

Purpose

- Sets a **breakpoint** at a specified line number in your script.
- If no argument is given, lists all currently set breakpoints.

Execution tracing

The final feature of the debugger is execution tracing, available with the x command. The function _xtrace "toggles" execution tracing simply by assigning to the variable _trace the logical "not" of its current value, so that it alternates between 0 (off) and 1 (on). The preamble initializes it to 0.

CONCLUSION

In this chapter, I understood that debugging Bash scripts is not just about fixing errors but about really seeing how each line runs. By using tools like `set -x`, `set -v`, and traps like `DEBUG` and `ERR`, I can trace my script step by step and catch problems early.

I also learned about `bashdb`, a simple yet powerful custom debugger that lets me set breakpoints, step through code, and interactively check variables. This gives me much more control, just like debuggers in other programming languages.

Overall, these techniques make my scripts easier to understand, maintain, and fix — and they make me more confident as a shell programmer.