

Final

Dbx Beginners Guide

4003-341-70
Edward Dale
1/27/04

Table of Contents

1. PREFACE	3
1.1. WHAT IS DBX?	3
1.2. PURPOSE.....	3
1.3. PROBLEM REPORTING.....	3
1.4. INTENDED AUDIENCE.....	3
1.5. APPLICABILITY.....	3
1.6. DOCUMENT CONVENTIONS.....	3
2. STARTING DBX	3
2.1. THE .DBXRC FILE	3
2.2. EXECUTING DBX.....	4
2.3. EXITING DBX	4
3. DEBUGGING A PROGRAM	4
3.1. LOADING/UNLOADING A PROGRAM	4
3.2. BREAKPOINTS.....	6
3.3. RUNNING YOUR PROGRAM.....	6
3.3.1. <i>Setting Program Arguments</i>	7
3.3.2. <i>Executing the Program</i>	7
3.4. INTERACTING WITH YOUR PROGRAM	8
3.4.1. <i>Resuming Execution</i>	8
3.4.2. <i>Examining your Program</i>	9
3.5. GETTING HELP	10
4. A SAMPLE DEBUGGING SESSION	11
4.1. DEBUGGING SESSION.....	11
4.2. DEBUGGING SESSION EXPLAINED.....	11
5. REFERENCES	13
6. INDEX OF COMMANDS	13

1. Preface

1.1. What is dbx?

"dbx is an interactive, source-level, command-line debugging tool. You can use it to run a program in a controlled manner and to inspect the state of a stopped program. dbx gives you complete control of the dynamic execution of a program, including the collection of performance data."¹

1.2. Purpose

This is a beginner's guide to using dbx to debug C++ programs on the RIT CS machines. Because I am taking a beginner's approach, many advanced commands will be omitted. The reader is advised to refer to the resources at the end of the document should they wish to pursue a more in-depth understanding. This guide is also meant to instruct the user how to use dbx over a text terminal to facilitate remote debugging. Therefore, the dbx commands that require a GUI will also be ignored.

1.3. Problem Reporting

All of the information contained in this document is correct to the author's knowledge. If there are any omissions, mistakes, or improvements that can be made, please let the author know. The current author is Edward Dale and can be reached at erd4819@cs.rit.edu.

1.4. Intended Audience

This document is written with students and faculty at Rochester Institute of Technology in mind. The author assumes that the audience is familiar with the concepts of a debugger and can navigate around a Unix command-line.

1.5. Applicability

This version has been written and tested on Version 6.1 of the Dbx Debugger. No other versions are guaranteed to work with the instructions listed here.

1.6. Document Conventions

I will use the following conventions when referring to program input and output.

Fixed space	User-entered commands and filenames.
<i>Italics</i>	Parameters to a command
Bold	dbx output.
#	The user's shell prompt.
)	The dbx prompt.
[]	Commands within brackets are optional.

2. Starting dbx

2.1. The .dbxrc file

The .dbxrc file stores commands that are executed when dbx starts. Dbx looks for the .dbxrc file in the current directory first and, if it is not found, the user's home directory. If the .dbxrc file is not found in either of those places, dbx looks for the .dbxinit file in the same order.

¹ <http://cs.rit.edu/doc/Workshop>

The `.dbxrc` file is most often used to setup aliases, ksh functions and `dbxenv` variables, all outside of the scope of this document. However, one handy use of it for a beginner is to add the line:

```
dbxenv suppress_startup_message 5.0
```

to the top of the `.dbxrc` file. This will suppress the new features startup message that is shown everytime `dbx` is started.

2.2. Executing `dbx`

To start using `dbx`, just run it at the command-line by typing `dbx`. You can optionally specify the program you wish to debug by typing it at the end of the command-line. If the program is not in the current directory, the full path must be typed.

Command Highlight: <code>dbx</code>
Executes <code>dbx</code> from the command-line.
Syntax: <code>dbx [program]</code> Executes <code>dbx</code> and optionally loads <code>program</code> to be debugged.
Parameter Summary: <code>program</code> The program you wish to debug. The full path is required if the program is not in the current directory.

2.3. Exiting `dbx`

To exit `dbx`, use the `quit` or `exit` command; both will accomplish the same thing. If you are currently attached to a process, `dbx` will automatically detach you from the process before exiting.

3. Debugging a Program

To debug a program, you must pass the `-g` flag to the compiler. This will cause additional symbol table information to be generated for `dbx` to use. On the RIT CS machines, the `makemake` program adds this flag automatically to the `makefile` when created.

3.1. Loading/Unloading a Program

To begin debugging your program, you must first load it into the debugger. The easiest way to do this to pass the program name to `dbx` when executing it

If you have already entered `dbx`, the `debug` command allows you to load your program into the debugger. If your program has already generated a core file and you would like to debug that, `dbx` provides that functionality also.

Command Highlight: <code>debug</code>
Displays or changes the current program being debugged.
<p>Syntax:</p> <pre>debug Displays the name and arguments of the program being debugged. debug [-r] program Begin debugging the specified program. debug [-r] program core Begin debugging the specified program using the specified core.</pre> <p>Parameter Summary:</p> <pre>-r Causes dbx to retain all display, trace, when, and stop commands. Its default behavior is to call delete all and undisplay 0 when a program is loaded. program The program you wish to debug. core The path to the core file you wish to debug.</pre>

If your program is already running and you would like the debugger to jump in and take control, the `attach` command is what you need. Upon execution, `dbx` will take control of your program, pause its execution, and await your command, as if a breakpoint had been encountered.

Command Highlight: <code>attach</code>
This command attaches <code>dbx</code> to a running process.
<p>Syntax:</p> <pre>attach [-r] pid Attach dbx to a running process.</pre> <p>Parameter Summary:</p> <pre>-r Causes dbx to retain all display, trace, when, and stop commands. Its default behavior is to call delete all and undisplay 0 when a program is attached. pid The process ID of the program you are attaching to.</pre>

Once done debugging, all you need to do is exit the program. But if you have another program that you would like to debug or some special reason, you may want to detach the program manually. This is done with the `detach` command.

Command Highlight: <code>detach</code>
Releases the program being debugged from <code>dbx</code> 's control.
<p>Syntax:</p> <pre>detach Release the program being debugged from dbx's control.</pre> <p>Parameter Summary:</p> <pre>No parameters.</pre>

3.2. Breakpoints

Now that the program is loaded into the debugger, it is time to set breakpoints so that you can investigate what is going on during execution. Breakpoints will halt the execution of your program and drop you down to the `dbx` command-line for further instructions. To set one, use the `stop` command.

Command Highlight: <code>stop</code>
This command sets breakpoints.
Syntax: <code>stop at line</code> Stop execution at the line in the source code. <code>stop in function</code> Stop execution when function is called.
Parameter Summary: line A line number in the source code. function A function name.

These breakpoints will persist until they are removed using the `clear` command or the program is detached from the debugger.

Command Highlight: <code>clear</code>
This command clears breakpoints.
Syntax: <code>clear</code> Remove all breakpoints. <code>clear [filename:] line</code> Remove all breakpoints at line.
Parameter Summary: line A line number in the source code. filename A source code file.

3.3. Running your program

With breakpoints set, it is time to run the program to see what is going on under the hood. The program will execute like normal, but everytime a breakpoint is encountered, `dbx` will drop you down to it's command-line to allow you to use the commands in the next section. If your program expects input, you can enter it like usual at the prompts.

3.3.1. Setting Program Arguments

In a lot of cases, it is desirable to pass arguments to your program. Dbx provides this functionality in the `runargs` command. Any arguments that are set persist until the program is detached or dbx is exited. Any type of argument can be set, including standard input redirections, permitting you to automate the input as is possible outside the debugger.

Command Highlight: <code>runargs</code>
This command sets arguments to be passed to the program.
Syntax: <code>runargs arguments</code> Set the run arguments. <code>runargs</code> Remove all run arguments.
Parameter Summary: <code>arguments</code> The arguments that you want passed to the program. Can include input and output redirection.

3.3.2. Executing the Program

Initially, the `run` command is used to start execution. It will start your program just as it would start on the command-line with the arguments you supplied with `runargs` and the breakpoints created with `stop`.

If, for some reason, you run into a program and need to force your program to exit, the same CTRL-DEL and CTRL-Z commands will work when you are in the debugger.

Command Highlight: <code>run</code>
Runs the program with breakpoints using the arguments supplied by <code>runargs</code> .
Syntax: <code>run</code> Run the program with arguments and breakpoints
Parameter Summary: No parameters.

If you have made some changes based on a previous run and wish to do everything that you just did a minute ago, the `replay` command will save you a couple keystrokes. It will play back all of the commands you have typed since the last `run` or `debug` command. It also provides the ability to preclude a certain number of commands, in case you added breakpoints or something else that does not need to be replayed.

Command Highlight: <code>replay</code>
Replays the debugging commands you used since the last run.
Syntax: <code>replay [-number]</code> Replay the commands typed since the previous run or debug command.
Parameter Summary: <code>number</code> The number of commands not to replay.

3.4. Interacting with your Program

Running your program and having it stop exactly when you want is all fine and dandy, but the true beauty of a debugger is being able to examine what is going on with the program at any specific instant. In this section, I will cover what to do when that assertion fails or you hit a breakpoint.

3.4.1. Resuming Execution

Once stopped at a breakpoint, it is important to be able to start back up again at the same point and inch your way through the source code.

`Dbx` provides the standard debugger functions of being able to step into and over code. In `dbx`, the command for stepping into code is `step`. This is a great function until you accidentally are stepped into the bowels of the STL. When this occurs, look to the `up` parameter of `step` to get you out of there.

<i>Command Highlight: step</i>
Step through source code (stepping into calls).
Syntax: <code>step</code> Step one line through the source code (into calls). <code>step [-n]</code> Step n lines through the source code (into calls). <code>step up</code> Step to the code that called execution to the current point.
Parameter Summary: n The number of steps to take.

When you know that a certain piece of code works or you know that you do not want to discover how the `cin` function works, stepping over code is handy. `Dbx` implements this with the `next` command.

<i>Command Highlight: next</i>
Step through source code (stepping over calls).
Syntax: <code>next</code> Step one line through the source code (over calls). <code>next [-n]</code> Step n lines through the source code (over calls).
Parameter Summary: n The number of steps to take.

Once finished debugging a certain piece of code, it is useful to be able to resume execution from the current point. `Dbx` allows you to do this with the `cont` command. `cont` also allows you to start at an arbitrary point. It is just like the `run` command except it starts you at the current point. Breakpoints are still obeyed.

Command Highlight: <code>cont</code>
Continue execution of the program.
Syntax: <code>cont</code> Continue execution of the program from the current line. <code>cont at line</code> Continue execution of the program from the specified line.
Parameter Summary: <code>line</code> Line to begin execution at.

3.4.2. Examining your Program

While your program is at a breakpoint, you have free reign to examine variables and expressions and call functions to see what exactly is going on. The strictly technical way that the debugger displays the data is often useful to see where exactly you might be going wrong. It is important to keep scope in mind when at this point, as the debugger will follow strict scope rules.

For looking at a variable's value or evaluating an expression once, the `print` command will suffice. It will display the value of the variable/expression immediately after your command and then allow you to enter further commands.

Command Highlight: <code>print</code>
Evaluates an expression or variable and displays it
Syntax: <code>print expression variable</code> Displays the evaluation of the expression/variable.
Parameter Summary: <code>expression</code> An expression to be evaluated <code>variable</code> A variable to be printed.

Another handy way to observe variables and expressions is to `display` them. When variables are `displayed`, every time the `dbx` prompt is showed, the value of that variable is showed also. It is essentially a persistent `print`, which is a great tool if you have something you want to keep an eye on while stepping through your code.

Command Highlight: <code>display</code>
Persistently print the value of a variable or expression
Syntax: <code>display expression variable</code> Persistently displays the evaluation of the expression/variable.
Parameter Summary: <code>expression</code> An expression to be evaluated <code>variable</code> A variable to be printed.

A great way to get information from your program is to actually ask it for the information. This is accomplished by calling procedures with the `call` command. If the procedure returns a value, it can be then be viewed using the `print` command detailed above. This method is also very useful during the debugging process to get your program to break by simulating what would happen in a particular circumstance.

<i>Command Highlight:</i> <code>call</code>
Call a procedure
Syntax: <code>call procedure([parameters])</code> Calls procedure with optional parameters.
Parameter Summary: procedure The procedure to be called parameters Parameters to optionally pass to the procedure.

3.5. Getting Help

This guide will hopefully be a handy guide to debugging, but if you are in the middle of a hard-core debugging session, you are not going to want to dig this out to find the exact syntax of a command. That is where `dbx`'s built-in help commands come into play.

If, at any point, you do find yourself looking for a command, but no knowing the name of it, the `commands` command is what you need. It will list the all built-in commands and a one-line description.

<i>Command Highlight:</i> <code>commands</code>
Lists all built-in commands with a one-line description.
Syntax: <code>commands</code> Lists all built-in commands with a one-line description.
Parameter Summary: No parameters.

Once you have found the command you needed, you are going to want the syntax of it. Using the `help` command, you can not only get the exact usage of a command, but also a categorized list of commands and other help topics. Note that the categorized list of commands does not include descriptions.

<i>Command Highlight: help</i>	
Provides help on a range of topics	
Syntax:	
<code>help</code>	Prints a categorized list of all built-in commands.
<code>help command</code>	Prints the usage of command.
Parameter Summary:	
<code>command</code>	The command to get help on.

4. A Sample Debugging Session

Here, I will present a full debugging session. Keep in mind the conventions mentioned in section 1.6 so you can tell what I am typing and what is being returned by `dbx`.

4.1. Debugging Session

<i>Debugging Session Output</i>	
1	<code>#make</code>
2	<code>CC -g -xsb -c Account.C</code>
3	<code>CC -g -xsb -c Bank.C</code>
4	<code>CC -g -xsb -c Owner.C</code>
5	<code>CC -g -xsb -o BankSim BankSim.o Account.o Bank.o Owner.o</code>
6	<code>#dbx BankSim</code>
7	Reading BankSim
8	Reading ld.so.1
9	Reading libCrun.so.1
10	Reading libm.so.1
11	Reading libw.so.1
12	Reading libc.so.1
13	Reading libdl.so.1
14	Reading libc_psr.so.1
15	<code>(/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) runargs < input.1</code>
16	<code>(/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) run</code>
17	Running: BankSim < input.1
18	(process id 11130)
19	Can't process check - account does not exist 23532
20	---Normal Program Output---
21	execution completed, exit code is 0
22	<code>(/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) stop at BankSim.C:28</code>
23	(2) stop at "BankSim.C":28
24	<code>(/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) run</code>
25	Running: BankSim < input.1
26	(process id 11131)
27	stopped in main at line 28 in file "BankSim.C"
28	28 Bank theBank;
29	<code>(/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) next</code>
30	stopped in main at line 29 in file "BankSim.C"

```

31 29 theBank.processInput();
32 (/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) print theBank
33 theBank = {
34   accountTypeCodeArray = (-4262104, 0, 0, 0)
35   accounts = {
36     __buffer_size = 4U
37     __buffer_list = {
38       __data_ = 0xb8e10
39     }
40   }
41 }
42 (/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) detach
43 detaching from process 11131
44 (/opt/SUNWspro/bin/./WS6U1/bin/sparcv9/dbx) quit
45 #

```

4.2. Debugging Session Explained

Lines 1-5

This is where the program (henceforth, BankSim) gets compiled. Notice the `-g` flag being passed to the compiler. As mentioned in section 3, this adds extra symbol information to the executable.

Lines 6-14

`dbx` is invoked on line 6. The remaining lines are `dbx` processing BankSim and getting ready to be able to execute it.

Line 15

Runtime arguments are shown. I am using standard input redirection here.

Lines 16-21

BankSim is executed on line 17 and runs through until line 20. Between those lines is all of the standard output that BankSim produces. On line 21, the exit code is displayed. This is handy in case your program uses exit codes to transmit information.

Lines 22-23

Here, a breakpoint is set in BankSim.C at line 28.

Lines 24-28

I run BankSim again and it stops as soon as it hits the breakpoint and returns me to the prompt.

Lines 29-31

Using the `next` command, I step over the current line to the next line.

Lines 32-41

Being interested in what `theBank` was initialized to, I use the `print` command to view its contents. Noticed the exploded view of the complex object `theBank`. Using this, it is much easier to see exactly what is going on. I can assume that the first value in `accountTypeCodeArray` was not initialized to zero, something that I may want to fix.

Lines 42-43

I am done here, so I `detach`. This is a superfluous command because I am about to quit the program in the next line.

Lines 44-45

Here, I quit `dbx` and am returned to my shell prompt.

5. References

This is a list of the resources that I used to create this document and other places that one might get help if they are having trouble or would like to gain a more advanced knowledge of `dbx` and debugging.

- <http://www.cs.rit.edu/doc/Workshop>
Forte Developer 6 Documentation Index
- <http://www.physics.utah.edu/~p573/hamlet/lessons/dbx/dbx/>
A similar document written by a Physics professor at Utah University
- The UNIX man page for `dbx`
`# man dbx`

6. Index of Commands

<code>attach</code> , 5	<code>help</code> , 11
<code>call</code> , 10	<code>next</code> , 8
<code>clear</code> , 6	<code>print</code> , 9
<code>commands</code> , 10	<code>quit</code> , 4
<code>cont</code> , 9	<code>replay</code> , 7
<code>dbx</code> , 4	<code>run</code> , 7
<code>debug</code> , 4, 5	<code>runargs</code> , 7
<code>detach</code> , 5	<code>step</code> , 8
<code>display</code> , 9	<code>stop</code> , 6
<code>exit</code> , 4	