DBX PRIMER

David Clarke

March, 2003

I. Overview

DBX is an interactive debugging environment that allows run-time interaction with your program for the purpose of uncovering programming bugs. DBX is generic in the sense that some version or other is available on all UNIX platforms. In addition, most platforms offer a window-based debugger based on DBX in which one can have separate portals to the source code, values of various variables being monitored, the "break-points", *etc.* As these window-based debuggers are different for every system and change with the whim of the vendor, this Primer will introduce DBX only.

II. Compiling your program for DBX

Under SOLARIS, the command

`f77 -g -C -ftrap=common` *program*

prepares your program for the debugger (`-g` option), flags all overflows, divides by zero, and invalid operations ($\sqrt{x < 0}$) (`-ftrap=common` option), and checks to make sure all arrays stay in bounds (*i.e.*, that you don't try to access `arr(11)` when `arr` is declared with only ten elements) (`-C` option).

Once compiled, type

`dbx a.out`

to put you into the DBX environment. After a header of about 30 or so relatively useless messages, you will find a ridiculously long prompt after which you are to enter your DBX commands.

III. DBX syntax

1. `list` n`,`m

Lists lines *n, m* in the current module, *i.e.*, the module in which execution has paused. When DBX is first fired up, you are paused just before the first line of the main program. You need to know line numbers for setting *breakpoints*, namely locations in the program where you wish to pause execution so you can probe the values of various variables, and these line numbers appear on the far left of the `list` listing.

2. `stop in` *modulename*

Sets a breakpoint right at the top of module *modulename*.

3. `stop at` n

Sets a break point at line n, where n is the left-most number of the listing generated by `list`. If line n is not an executable line (*e.g.*, a comment), the breakpoint is set to the first executable line following.

4. `stop at` n `-if` *expression*

This is a conditional breakpoint, and applies equally well to the `stop in` command. Execution is stopped at line n only if the expression indicated is true. This is particularly useful for stopping inside a long do-loop. For example, if I wanted to probe the value for `d(i,j)` when $j = 67$ and $i = 33$ in the following coding snippet:

```
210       do j=1,jmax
211         do i=1,imax
212           d(i,j) = d(i,j) - (mflx(i+1) - mflx(i)) * dt / vol(i)
213           e(i,j) = e(i,j) - (eflx(i+1) - eflx(i)) * dt / vol(i)
214         enddo
215       enddo
```

I would issue the following commands:

```
stop at 211 -if j==67
cont
stop at 212 -if i==33
cont
```

The minus sign in front of the `if` and double equals sign (both absent in the original DBX) are examples of what happens when computer scientists get their hands on perfectly good software.

5. `status`

Lists all current breakpoints. If you are currently stopped at a breakpoint, it will have an asterisk to the far left of the `status` listing. The breakpoint numbers appearing on the left of the list are how you refer to breakpoints when you want to, for example, delete them (next paragraph).

6. `delete` n m

Deletes breakpoints n and m, where n and m are the breakpoint numbers on the list generated by `status`.

7. `run`

Begins execution of the program from the very beginning, and continues until the first breakpoint is reached. Execution is stopped just before the line of the breakpoint is executed.

8. `cont`

Continues execution from the current location, and up to but not including the next breakpoint.

9. `next`

Executes the next line of the current module, then stops. If the next line is a call to a subroutine, that entire subroutine is executed so that execution is paused in the same module.

10. `step`

Executes the next line encountered. If the next line is a subroutine call, then execution is broken at the first line of the subroutine, and you are now paused inside the subroutine, rather than the calling module.

11. `where`

If you lose track of where you are...

12. `print` *variablename*

Prints the value of the variable on the screen. If *variablename* is an array, all elements of the array are printed (or perhaps the first hundred values, I forget). To print just a portion of *variablename* if it is an array, type

`print` *variablename*`(`m_1`:`m_2`,`n_1`:`n_2`,...)`

where a specific range is specified for every dimension of the array.

13. `assign` *expression*

Sometimes, you want to see what would happen if the value of a variable were to change. The `assign` statement allows you to do this. For example, if `iter` were the variable keeping track of the number of iterations, and you wanted to test your escape trap for when `iter` exceeds the maximum allowed value (say 100), then you might type:

```
assign iter=101
cont
```

14. `Control-C`

`Control-C` is captured by DBX, and stops the execution of your program without exiting DBX. In fact, it gives you a DBX prompt, which you can use to find out where you are, set more breakpoints, probe variable values, or quit.

15. `quit`

Exits from DBX.