# A *DBX* Primer

David A. Clarke

Saint Mary's University, Halifax NS, Canada

david.clarke@smu.ca

March 2003; revised 1/11, 5/12

# Contents

# Preface

This primer gives a quick guide to *DBX*, a line debugger developed and distributed by SUN Microsystems™. It has since been ported to a variety of other platforms and has been used as the base of several windows-based debuggers. As a *FORTRAN* programmer, I have used a debugger of one form or another since the late 1980s. In my view, while there are more powerful debugging environments, *DBX* is one of the easiest debuggers to use and one of the most effective I have come across.

These notes were originally written for my students, particularly those in my undergraduate computational physics course. I make them available to anyone else to use, distribute, and modify as needed so long as they remain in the public domain and are passed on to others free of charge.

David Clarke
Saint Mary's University
January, 2011; revised May, 2012

*Primers by David Clarke:*

1. A *FORTRAN* Primer
2. A *UNIX* Primer
3. A DBX (debugger) Primer
4. A Primer on Tensor Calculus
5. A Primer on Magnetohydrodynamics
6. A Primer on *ZEUS-3D*
7. Eclipse Photography for Mortals

I also give a link to David R. Wilkins' excellent primer Getting Started with LaTeX, in which I have added a few sections on adding figures, colour, and HTML links.

# A *DBX* Primer

## 1 Overview

*DBX* is an interactive debugging environment that allows run-time interaction with your program for the purpose of uncovering programming bugs. Running your program through *DBX* or any debugger allows you to step through the program line by line, set breakpoints, probe variable values, and even change some variable values in case you are wondering "what would happen if ...".

Learning to use a debugger can take a bit of time but, without exception, those who learn to use them never look back. There is nothing more frustrating and less productive in code development than having to recompile and rerun a program every time you add a new write statement in your hunt for the first place a certain variable runs afoul. Using write statements to debug a code *vs.* a proper debugger is the difference between carrying your water from a well in a leaky bucket, and turning on the kitchen faucet.

## 2 Compiling your program for *DBX*

*DBX* was developed by SUN Microsystems, and while it has been ported to many *UNIX* systems (*e.g.*, `venus` is an Intel system running its flavour of *UNIX*: `CentOS 5`), it doesn't seem to work with non-SUN compilers. The universal compiler option to prepare a program for a debugger is `-g`, though different compilers do different things to a code when the `-g` flag is set and, on `venus` at least, *DBX* cannot analyse an executable unless it is prepared by the SUN compiler `f90` (which `venus` supports).

Thus, to prepare your program for *DBX*, you must compile your program using:

`f90 -g -C -ftrap=common` *program*

In addition to the `-g` flag, the `-C` flag 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), while the `-ftrap=common` checks for overflows, divides by zero, and invalid operations ($\sqrt{x < 0}$). While your code is in development, it is always a good idea to have these flags set.

Once compiled, type:

`dbx a.out`

to put you into the *DBX* environment. After several welcoming and introductory messages, you will find the cursor waiting for you at the *DBX* prompt, `(dbx)`, after which you are to enter your *DBX* commands.

Incidentally, *DBX* was initially designed for use with `C++`, and compiling your `C++` program with `cc -g` *program* (`cc` is SUN's `C`-compiler) will give you an executable suitable for *DBX*. Finally, a `gnu` debugger that works with `gfortran` is `gdb`, which you can download free of charge with `gfortran`, and which bears some similarity to *DBX*.

# 3 Syntax

The following subsections give the syntax and a brief description of a subset of *DBX* commands that I have found allows me to do pretty well all I need to do in a line-command debugger. Commands are given roughly in the order that a novice user might need them. Still, it is probably best to peruse all the commands quickly before launching into your first *DBX* session to know what is available. An alphabetical listing of the commands discussed and their subsection number follows.

| command | § | command | § |
|---|---|---|---|
| `assign` *expression* | 3.16 | `quit` | 3.20 |
| `cont` | 3.9 | `run` | 3.8 |
| `Control-C` | 3.12 | `status` | 3.6 |
| `delete` *n m* | 3.7 | `step` | 3.11 |
| `display` *variablename* | 3.14 | `stop at` *n* | 3.4 |
| `func` *modulename* | 3.2 | `stop at` *n* `-if` *expression* | 3.5 |
| `help` *commandname* | 3.19 | `stop in` *modulename* | 3.3 |
| `list` *n*`,`*m* | 3.1 | `undisplay` *variablename* | 3.15 |
| `next` | 3.10 | `where` | 3.18 |
| `print` *variablename* | 3.13 | `whereis` *variablename* | 3.17 |

## 3.1 `list` *n*`,`*m*

Lists lines *n, m* in the current module, *i.e.*, the module in which execution has paused or to which `func` has redirected the scope. 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.

## 3.2 `func` *modulename*

Shifts the "scope" to that of the named module (see `print` for a brief discussion on "scope"). By default, the scope is that of the module in which execution is stopped. Changing the scope to a different module means, among other things, that `list` will list the contents of the new module, variables within the scope of the new module can be probed, *etc.* Note that issuing the `func` command does not cause execution to advance to the named module; it simply redirects the scope.

## 3.3 `stop in` *modulename*

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

## 3.4 `stop at` *n*

Sets a breakpoint at line $n$ of the "current module", where $n$ is the left-most number of the listing generated by `list`. The current module can be changed with `func`, if desired. If line $n$ is not an executable line (*e.g.*, a comment), the breakpoint is set to the first executable line following.

## 3.5 `stop at` *n* `-if` *expression*

This is a conditional breakpoint, and applies equally well to the `stop in` command. Execution is stopped at line $n$ of the current module 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
```

Note the "double equal sign" to indicate equality; why "`=`" isn't enough, I couldn't tell you. Other legal operators include greater than (`>`), greater than or equal (`>=`), less than (`<`), and less than or equal (`<=`). Thus,

```
stop at 212 -if i>=33
```

would stop at line 212 when `i=33`, and then every value of `i` thereafter.

## 3.6 `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 (see `delete`).

## 3.7 `delete` *n m*

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

## 3.8 `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. See also `cont`.

## 3.9 `cont`

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

## 3.10 `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. See also `step`.

## 3.11 `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. See also `next`.

## 3.12 `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`.

## 3.13 `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.

*Note on "scope"*: Variables that can be accessed (*e.g.*, whose values can be printed) from within a programming module (subroutine or function) must be within that module's *scope*, a subset of the program visible from within the module. Generally, to be in a module's

scope, a variable must be accessed by that module (*e.g.*, used in an assignment statement, passed to another module, *etc.*.), though there are numerous exceptions and I have yet to determine the definitive criteria which define what the scope of a module is.

The following session shows what can happen when a requested variable is not in scope, and what can be done to access the variable:

```
(dbx) print nhy
dbx: "nhy" is not defined in the scope ‘xdzeus36‘srcstep.f‘srcstep‘
(dbx) whereis nhy
Common variable: ‘xdzeus36‘zeus3d.f‘MAIN‘nhy
(dbx) print ‘xdzeus36‘zeus3d.f‘MAIN‘nhy
nhy = 0
(dbx)
```

Alternatively, one can change the scope using `func`, and then probe for the variable value:

```
(dbx) print nhy
dbx: "nhy" is not defined in the scope ‘xdzeus36‘srcstep.f‘srcstep‘
(dbx) func zeus3d
(dbx) print nhy
nhy = 0
(dbx)
```

Annoyingly, not all installations of *DBX* are the same. Under the old SUN Sparc machines, all common variables were considered in scope; on `venus`, this isn't the case. Even worse, sometimes `whereis` gets confused, and the answer it gives is insufficient to access the desired variable. In these cases, I find all I can do is quit *DBX*, put into the module a dummy statement just to put the variable I want to access in the module's scope [*e.g.*, something innocuous like `d(1,1,1)=d(1,1,1)`], recompile, and then resume my debugging session.

See also `display`.

## 3.14 `display` *variablename*

Like print, the value of the variable is printed on the screen. However, displayed variables are echoed at every subsequent breakpoint with their updated values shown. This list can get long, so it should be used sparingly. See also `print` and `undisplay`. In particular, see `print` for what to do if variables are declared "not in scope".

## 3.15 `undisplay` *variablename*

Removes a variable from a display list. See also `display`.

## 3.16 `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
```

## 3.17 `whereis` *variablename*

Indicates the module within which the specified variable is "in scope". See `print` for an example.

## 3.18 `where`

If you lose track of where you are...

## 3.19 `help` *commandname*

Provides on-line help for the specified command; *e.g.*, `help print`

## 3.20 `quit`

Exits from *DBX*.