NAME

dbx − debugger

SYNOPSIS

dbx [**−r**] [**−i**] [**−I** *dir*] [*objfile* [*coredump*]]

DESCRIPTION

Dbx is a tool for source level debugging and execution of programs under UNIX. The *objfile* is an object file produced by a compiler with the appropriate flag (usually ‘‘−g’’) specified to produce symbol information in the object file. Currently, *cc*(1), *f77*(1), and the DEC Western Research Laboratory Modula-2 compiler, *mod*(l), produce the appropriate source information. The machine level facilities of *dbx* can be used on any program.

The object file contains a symbol table that includes the name of the all the source files translated by the compiler to create it. These files are available for perusal while using the debugger.

If a file named ‘‘core’’ exists in the current directory or a *coredump* file is specified, *dbx* can be used to examine the state of the program when it faulted.

If the file ‘‘.dbxinit’’ exists in the current directory then the debugger commands in it are executed. *Dbx* also checks for a ‘‘.dbxinit’’ in the user’s home directory if there isn’t one in the current directory.

The command line options and their meanings are:

−r Execute *objfile* immediately. If it terminates successfully *dbx* exits. Otherwise the reason for termination will be reported and the user offered the option of entering the debugger or letting the program fault. *Dbx* will read from ‘‘/dev/tty’’ when **−r** is specified and standard input is not a terminal.

−i Force *dbx* to act as though standard input is a terminal.

−I *dir* Add *dir* to the list of directories that are searched when looking for a source file. Normally *dbx* looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the **use** command.

Unless **−r** is specified, *dbx* just prompts and waits for a command.

Execution and Tracing Commands

run [*args*] [< *filename*] [> *filename*]
rerun [*args*] [< *filename*] [> *filename*]

Start executing *objfile*, passing *args* as command line arguments; < or > can be used to redirect input or output in the usual manner. When **rerun** is used without any arguments the previous argument list is passed to the program; otherwise it is identical to **run**. If *objfile* has been written since the last time the symbolic information was read in, *dbx* will read in the new information.

trace [**in** *procedure/function*] [**if** *condition*]
trace *source-line-number* [**if** *condition*]
trace *procedure/function* [**in** *procedure/function*] [**if** *condition*]
trace *expression* **at** *source-line-number* [**if** *condition*]
trace *variable* [**in** *procedure/function*] [**if** *condition*]

Have tracing information printed when the program is executed. A number is associated with the command that is used to turn the tracing off (see the **delete** command).

The first argument describes what is to be traced. If it is a *source-line-number*, then the line is printed immediately prior to being executed. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, e.g. "mumble.p":17.

If the argument is a procedure or function name then every time it is called, information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted, and if it's a function then the value it is returning is also printed.

If the argument is an *expression* with an **at** clause then the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable then the name and value of the variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified then all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause "**in** *procedure/function*" restricts tracing information to be printed only while executing inside the given procedure or function.

Condition is a boolean expression and is evaluated prior to printing the tracing information; if it is false then the information is not printed.

stop if *condition*
stop at *source-line-number* [**if** *condition*]
stop in *procedure/function* [**if** *condition*]
stop *variable* [**if** *condition*]

Stop execution when the given line is reached, procedure or function called, variable changed, or condition true.

status [> *filename*]

Print out the currently active **trace** and **stop** commands.

delete *command-number ...*

The traces or stops corresponding to the given numbers are removed. The numbers associated with traces and stops are printed by the **status** command.

catch *number*
ignore *number*

Start or stop trapping signal *number* before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. Initially all signals are trapped except SIGCONT, SIGCHILD, SIGALRM and SIGKILL.

cont Continue execution from where it stopped. Execution cannot be continued if the process has "finished", that is, called the standard procedure "exit". *Dbx* does not allow the process to exit, thereby letting the user to examine the program state.

step Execute one source line.

next Execute up to the next source line. The difference between this and **step** is that if the line contains a call to a procedure or function the **step** command will stop at the beginning of that block, while the **next** command will not.

return [*procedure*]

Continue until a return to *procedure* is executed, or until the current procedure returns if none is specified.

Displaying and Naming Data

print *expression* [**,** *expression ...*]

Print out the values of the expressions. Array expressions are always subscripted by brackets ("[ ]"). Variables having the same identifier as one in the current block may be referenced as "*block-name* **.** *variable*". The field reference operator (".") can be used with pointers as well as records, making the C operator "->" unnecessary (although it is supported). The construct *expression* \ *typename* can be used to print the *expression* out in the format of the type named *typename*.

whatis *name*
Print the declaration of the given name, which may be qualified with block names as above.

which *identifier*
Print the full qualification of the given identifer, i.e. the outer blocks that the identifier is associated with.

whereis *identifier*
Print the full qualification of all the symbols whose name matches the given identifier. The order in which the symbols are printed is not meaningful.

assign *variable* = *expression*
set *variable* = *expression*
Assign the value of the expression to the variable.

call *procedure(parameters)*
Execute the object code associated with the named procedure or function. Currently, calls to a procedure with a variable number of arguments are not possible. Also, string parameters are not passed properly for C.

where Print out a list of the active procedures and function.

dump [> *filename*]
Print the names and values of all active variables.

up [*count*]
down [*count*]
Move the current function, which is used for resolving names, up or down the stack *count* levels. The default *count* is 1.

Accessing Source Files

edit [*filename*]
edit *procedure/function-name*
Invoke an editor on *filename* or the current source file if none is specified. If a *procedure* or *function* name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. The default can be overridden by setting the environment variable EDITOR to the name of the desired editor.

file [*filename*]
Change the current source file name to *filename*. If none is specified then the current source file name is printed.

func [*procedure/function*]
Change the current function. If none is specified then print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

list [*source-line-number* [**,** *source-line-number*]]
list *procedure/function*
List the lines in the current source file from the first line number to the second inclusive. If no lines are specified, the next 10 lines are listed. If the name of a procedure or function is given lines *n-k* to *n+k* are listed where *n* is the first statement in the procedure or function and *k* is small.

use *directory-list*
Set the list of directories to be searched when looking for source files.

Machine Level Commands

tracei [*address*] [**if** *cond*]
tracei [*variable*] [**at** *address*] [**if** *cond*]
stopi [*address*] [**if** *cond*]
stopi [**at**] [*address*] [**if** *cond*]
Turn on tracing or set a stop using a machine instruction address.

stepi

nexti Single step as in **step** or **next**, but do a single instruction rather than source line.

address **,***address***/** [*mode*]
[*address*] **/** [*count*] [*mode*]
Print the contents of memory starting at the first *address* and continuing up to the second *address* or until *count* items are printed. If no address is specified, the address following the one printed most recently is used. The *mode* specifies how memory is to be printed; if it is omitted the previous mode specified is used. The initial mode is ‘‘X’’. The following modes are supported:

i	print the machine instruction
d	print a short word in decimal
D	print a long word in decimal
o	print a short word in octal
O	print a long word in octal
x	print a short word in hexadecimal
X	print a long word in hexadecimal
b	print a byte in octal
c	print a byte as a character
s	print a string of characters terminated by a null byte
f	print a single precision real number
g	print a double precision real number

Symbolic addresses are specified by preceding the name with an ‘‘&’’. Registers are denoted by ‘‘$rN’’ where N is the number of the register. Addresses may be expressions made up of other addresses and the operators ‘‘+’’, ‘‘-’’, and indirection (unary ‘‘*’’).

Miscellaneous Commands

sh *command-line*
Pass the command line to the shell for execution. The SHELL environment variable determines which shell is used.

alias *new-command-name old-command-name*
Respond to *new-command-name* as though it were *old-command-name*.

help Print out a synopsis of *dbx* commands.

gripe Invoke a mail program to send a message to the person in charge of *dbx*.

source *filename*
Read *dbx* commands from the given *filename*.

quit Exit *dbx*.

FILES

a.out	object file
.dbxinit	initial commands

SEE ALSO

cc(1), f77(1), mod(l)

COMMENTS

Non-local gotos can cause some trace/stops to be missed. Most of the command names are too long. The alias facility helps, but is really quite weak. A *csh*-like history capability would improve the situation. But then, who wants to duplicate the c-shell in a debugger?

Dbx suffers from the same ‘‘multiple include’’ malady as does sdb. If you have a program consisting of a number of object files and each is built from source files that include header files, the symbolic information for the header files is replicated in each object file. Since about one debugger start-up is done for each link, having the linker (ld) re-organize the symbol information won’t save much time, though it would reduce some of the disk space used. The problem is an artifact of the unrestricted semantics of #include’s in C; for example an include file can contain static declarations that are separate entities for each file in which they are included.