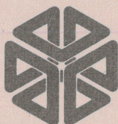
Tutorial:

Learning to Debug with edge

C Edition



SiliconGraphics
Computer Systems

Document number: 007-0903-020

Learning to Debug with *edge*

C Edition

Version 2.0

Document Number 007-0903-020

Technical Publications:

Amy B. W. Smith
Kevin B. Walsh
Beverly White
Diane Wilford

Engineering:

Greg Boyd
Jeff Doughty
Deb Ryan
Jim Terhorst

© Copyright 1988, Silicon Graphics, Inc. - All rights reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc., and is protected by Federal copyright law. The contents of this document may not be disclosed to third parties, copied or duplicated in any form, in whole or in part, without the express written permission of Silicon Graphics, Inc.

U.S. Government Limited Rights

Use, duplication or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (b) (2) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Contractor/manufacturer is Silicon Graphics Inc., 2011 Stierlin Road, Mountain View, CA 94039-7311.

Learning to Debug with *edge*
C Edition
Version 2.0
Document Number 007-0903-020

Silicon Graphics, Inc.
Mountain View, California

UNIX is a trademark of AT&T Bell Laboratories.

Contents

To the Reader i

1. What is edge? 1
- Preparing a Program for Use under *edge* 2
- Using Makefile to Set Up the edge Tutorial 3
  - Bug #1 5
  - Bug #2 10
- Summary of Basic Commands 13

2. More Elusive Bugs 15
- Understanding Some Advanced Commands 15
- Using the Advanced Commands 17
  - Bug #3 17
  - Bug #4 23
  - Bug #5 27
- Summary of Advanced Commands 31

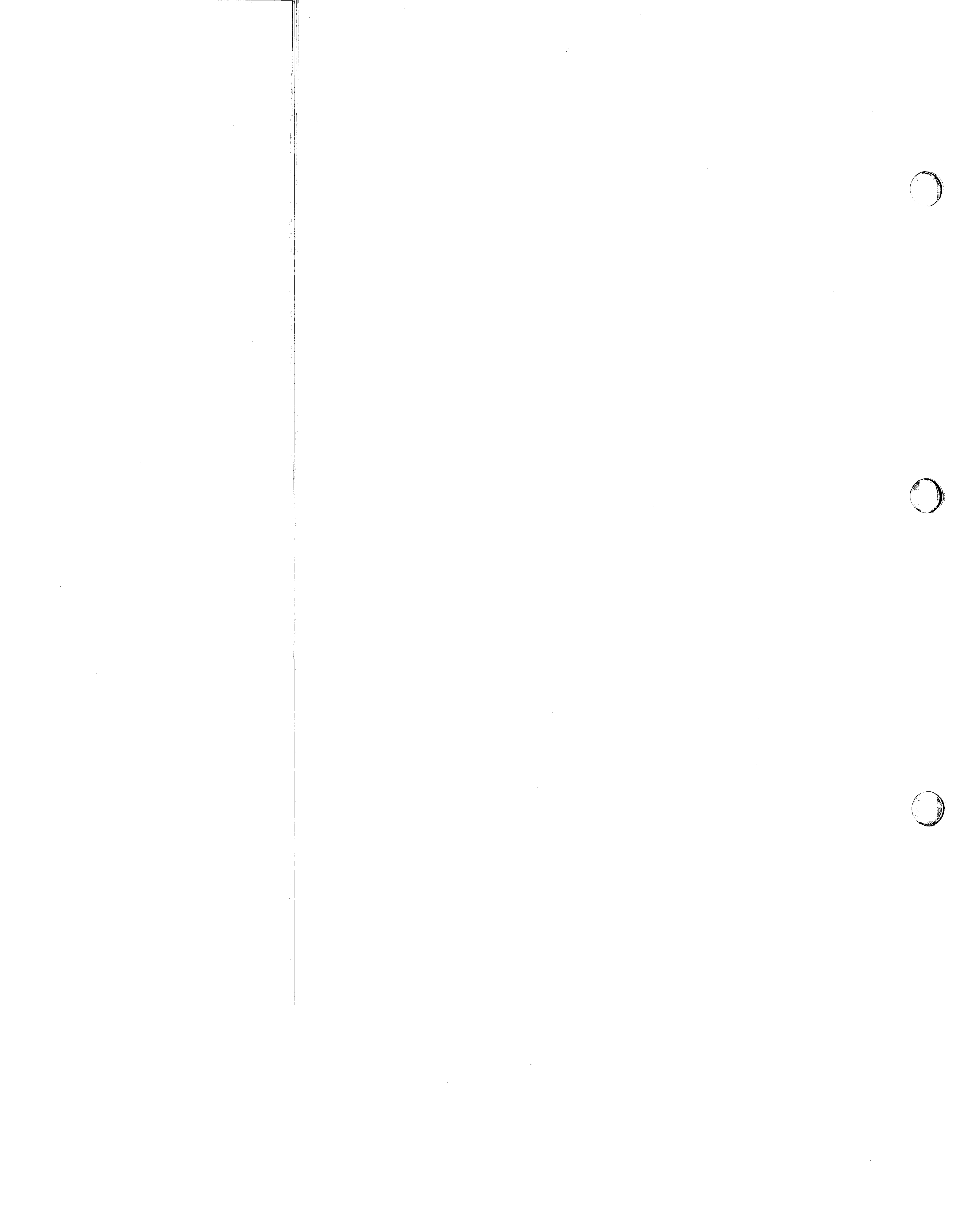
3. On Your Own 33
- Using *edge* to Debug Graphics Programs 33
- The Debugging Process 34
- Summary of *edge* Commands 35
  - Textual Commands 35
  - Choices on the Command Menu 36
  - Choices on the Pop-up Menu 37
  - *vi* Search Commands 38
- Where to Find Additional Information 38

To the Reader

This tutorial is designed for C programmers with little or no experience using the Silicon Graphics, Inc. graphical debugger, *edge*. After only one or two hours with this tutorial you will be able to use *edge* to debug your programs more quickly and efficiently. You will learn:

- how to prepare a program for debugging under *edge*
- how to use the *edge* interface
- how to use both basic and advanced debugging commands to debug sample programs
- general rules to help you debug your own programs

To use this tutorial you need a basic understanding of UNIX and the *vi* text editor. Read *Getting Started with the IRIS-4D Series Workstation* if you need to learn or review this information.



1. What is edge?

edge is a window-based, graphical interface to *dbx*, a standard UNIX debugger. You can use *dbx* to find bugs in your executable files, and if those executable files are compiled using the *-g* compiler option, *dbx* can relate the executable code to the source code. Specifically, *dbx* lets you:

- stop your program at specified points to check current values
- trace variables as they change throughout your program
- step through functions one line at a time

The *edge* interface to *dbx* consists of three independent windows: the Command Window, the Source Window, and the User Window. You can use the Command Window to issue *dbx* commands manually; you can use the Source Window to view the source code as it executes; and you can use the User Window to monitor the program input/output (standard in and standard out) and error messages (standard error).

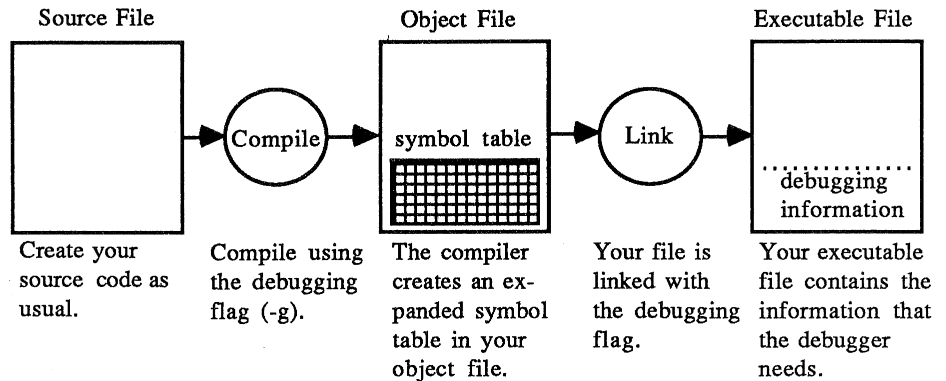
Because *edge* runs under the Silicon Graphics, Inc. window manager, *4Sight*, it is not always necessary to type in *dbx* commands. The most common *dbx* commands are mapped to menus in the Command Window and the Source Window. The window manager also allows you to select command input (e.g., program variables) via the mouse.

Another advantage of running *edge* under the *4Sight* window manager is that you can use *edge* to debug graphics programs that also run under the *4Sight* window manager. See Chapter 3 for more information about using *edge* with graphics programs.

Preparing a Program for Use under *edge*

You do not need to make any changes to your source code to run the code under *edge*. However, to take advantage of all the *edge* and *dbx* features, you should compile the program using the *-g* compiler option. The *-g* compiler option ensures that the final executable file contains an expanded symbol table. Using this table, *edge* and *dbx* can relate lines of machine code to lines of source code and display that source code as it executes in the source window.

In addition, when preparing an executable for use under *edge*, you should not optimize the code. Optimized code can be submitted to *edge*, however, because optimization rearranges the machine code, following the execution of such a program can be very difficult.



Using Makefile to Set Up the edge Tutorial

You will be working on a sample program called *sort.c*. During this session you will use nine basic *edge* commands to eliminate two bugs. Your IRIS should be booted and displaying the `IRIS login:` prompt. Log in as *tutor*, and change directories so that your current working directory is */usr/tutor/edge/C/src*. Type:

```
cd  /usr/tutor/edge/C/src
```

To set up the *edge* tutorial environment, type:

```
make
```

When the system prompt appears again, list the contents of this directory. Type:

```
ls
```

You see six file names: *Makefile, names.in, scrub, sort, sort.c*, and *sort.m*. The program *sort.c* reads the input file *names.in*, sorts it, and puts the results into an output file. The code that you will debug in this section processes command line arguments; the actual sorting is done by the C library routine, *qsort*. To briefly look over *sort.c*, type:

```
more  sort.c
```

Press <spacebar> to look at the next screenful; press <delete> to stop viewing the program and return to the system prompt.

Note: If you find any bugs, do not try to fix them!

When you feel comfortable with the structure of *sort.c*, return to the system prompt.

The *Makefile* in this directory helps you do the tutorial at your own pace, and lets you easily restore the directory so someone else can start fresh with the tutorial.

If you need to stop before you complete the tutorial, you can save your work and pick up where you left off later. To save your bug fixes, type:

```
make  save
```

When you want to resume the tutorial, return to the */usr/tutor/edge/C/src* directory and type:

```
make  restore
```

Finally, when you complete the tutorial, restore the directory so that someone else can use the tutorial. Type:

```
make  done
```

Now you are ready to tackle the first bug.

Bug #1

1. Compile and link *sort.c* using the *-g* flag, and name your executable file *sort.*

```
cc  -g  sort.c  -o  sort
```

2. Run your program using the input file *names.in*, and put the sorted results into a new output file called *names.out*.

```
sort  names.in  -o  names.out
```

You see this message:

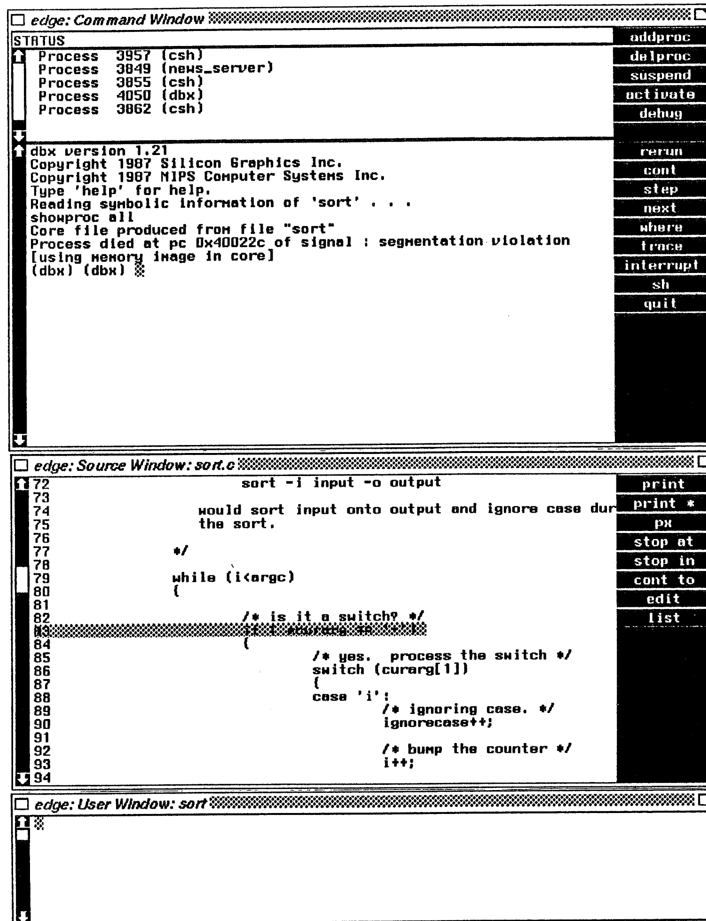
```
sort: Segmentation violation -- Core dumped
```

This means *sort* has a bug that causes a program fault. The "Segmentation violation" message usually means that there is a bad pointer reference in your code. "Core dumped" means that UNIX took the memory image of your program when it faulted, and put it into a file named *core*.

3. Submit *sort* to *edge*. Type:

```
edge  sort
```

The system displays the three *edge* windows shown on the next page.



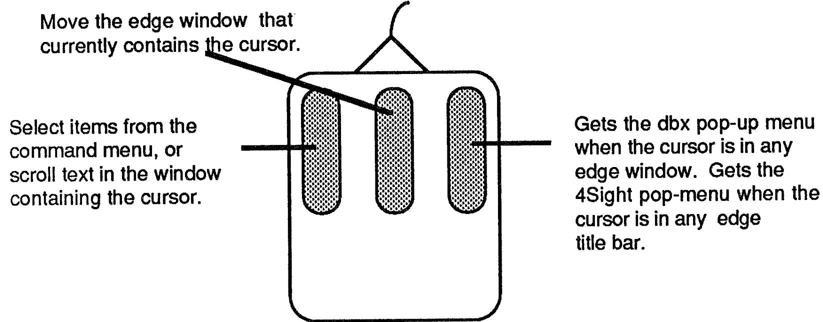
The top window, the Command Window, contains a command menu, a process list, and a *dbx* command processor. The top section of the Command Window the process list, lists all the processes associated with your login. The lower section of the Command Window, the *dbx* command processor, receives typed command to *edge*, and runs all of the standard *dbx* commands.

The middle window, the Source Window, lists the source code that you are currently debugging. You can scroll through the source code by placing your cursor over the "up" or "down" arrows of the scroll bars and clicking the left mouse button.

You can also scroll text by placing the cursor on the elevator block of the scroll bar, pressing and holding the left mouse button, and dragging the cursor up or down.

The bottom window, the User Window, displays the results you get when you run the program (standard in, standard out, and standard error).

To use the commands on a command menu, position the cursor over the menu item and press the left mouse button. If the command requires an object, you must highlight that object before you select the command. To highlight an object (e.g., a variable in the source code or a process listed in the top of the command window), position the cursor over the start of the object, press and hold the left mouse button, drag the cursor to the end of the object, and release the left mouse button.



If you look at the Source Window, you notice that line 83 is highlighted. This is the line at which the program faulted.

4. Get more information about the fault. Position your cursor over the word 'where' in the command menu, and press the left mouse button.

   In the Command Window, *edge* displays the message:

```
> 0 main(argc=4, argv=0x7fffdba4) ["sort.c":83, 0x4001d8]
```

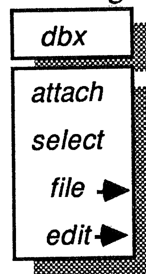
   Whenever a line of code that causes a program fault contains a variable, you should check its value.

5. Check the value of *curarg* (current argument), use the *print* command.

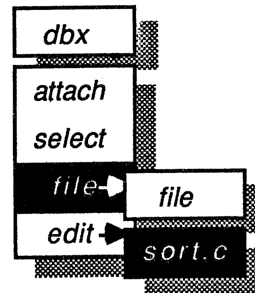
   To use *print*, first highlight the variable that you want to print, then select 'print' from the command menu. Highlight *curarg* by positioning your cursor between the asterisk (*) that precedes *curarg*, pressing and holding the left mouse button, and dragging the cursor over the rest of the word. When the entire word is highlighted, release the left button. Now use the left button to select 'print' from the command menu.

   Since the value of *curarg* is nil, you want to make sure that it was initialized. As you see in the Source Window, this never happened. *curarg* should have been initialized between lines 80 and 82.

6. Edit the source file to add initialization code for the program variable *curarg*. To edit your source file, position your cursor in any *edge* window, and press and hold the right mouse button. You see this menu:



   Move down the menu so that 'edit' is highlighted, then carefully slide your cursor to the right. You see a sub-menu that contains only one choice — 'sort.c'. Make sure it is highlighted (your cursor should be on top of it), then release the mouse button.



You see a red outline. Move the cursor down to the lower left-hand corner of your screen, and press and release the right mouse button. You have just created a new UNIX shell that is running the *vi* text editor on your source file, *sort.c*. (When your program consists of several source files, the 'edit' sub-menu contains all of them so you can access them easily.)

7. Add temporary line numbers to your file so that you can edit it exactly as this tutorial does. Tell *vi* to display line numbers, enter the command:

```
:set number
```

   It is important to perform recommeded edits exactly. This tutorial makes references to code by line number. If your edits change line count in ways we have not anticipated, you will find it difficult to complete the rest of the tutorial.

8. Edit the code so that lines 79-85 look like this:

```
79          while (i<argc)
80          {
81
82               /*  get the current argument  */
83               curarg  =  argv[i];
84
85               /* is it a switch? */
```

9. Save your edits and exit from *vi* as usual. When you do this, the new shell disappears.

```
:wq
```

10. Recompile sort.c using the *-g* option (there are still more bugs to find). Move the cursor to the Command Window and enter the command:

```
sh cc -g sort.c -o sort
```

You have successfully eliminated the first bug.

Bug #2

1. Run the program within the *edge* environment. Use the *run* command to run *sort* in *edge*, using *names.in* and *names.out*.

```
run  -o  names.out  names.in
```

   It is not necessary to specify the name of the program. *edge* assumes you want to run the program specified when you started. In addition, *edge* rereads the object code. In the Command Window, *edge* displays the message:

```
Process 6088 (sort) started.
Object hs been remade.  Re-reading symbolic information ..
Process  6088 (sort) terminated
Process  6089 (sort) started
Process  6089 (sort) finished
```

   In the User Window, *edge* displays the message:

```
sorting . . .
7 records sorted from input file names.in
   onto output file -o
```

   It seems that the file was sorted, but the output file was named *-o* rather than *names.out*. It's likely that there is a problem where the output file is assigned. Look for this code in the Source Window by scrolling through the text.

2. Scroll to line 105. To scroll through the text, move the cursor to the scroll bar at the left of the Source Window. Using the left mouse button, click on the arrows to scroll the text.

3. Set a breakpoint at the line in which the name of the output file is assigned. A breakpoint at a line makes *dbx* stop executing the program and display the line containing the breakpoint. To set a breakpoint, use the 'stop at' command. To use 'stop at', highlight the line of code (line 105) using the left mouse button, then select 'stop at' from the command menu.

4. Run *sort* again. If you have already run a program in *edge*, you can easily run it again with the same arguments by using the *rerun* command. Select 'rerun' from the command menu.

5. Line 105 contains the variable *argv[i]*. Check its value by highlighting it using the left mouse button, then selecting 'print' from the command menu.

   The value is *-o*. This is the argument which appears on the command line one position before the desired output file, *names.out*. This means that the dummy counter *i* has not been incremented properly.

6. To check that the next member of *argv[ ]* is actually *names.out*, at the `(dbx)` prompt in the Command Window, type:

   ```
   print  argv[i+1]
   ```

   The value is *names.out*, as it should be.

7. Edit *sort.c* by placing the cursor in any *edge* window, pressing the right mouse button, and selecting 'sort.c' from the rollover menu that is beneath the 'edit' choice.

8. Tell *vi* to display line numbers.

   ```
   :set number
   ```

9. Change your code so that lines 101-109 look like this:

```
101        /* the output file name follows */
102
103        /* increment past the switch.  The
104           next argument is the name of the
105           output file. */
106
107        i++;
108
109        if (i<argc)
```

10. Save your changes and exit from *vi.*

```
:wq
```

11. Exit from *edge* by selecting 'quit' from the command menu.

12. Recompile *sort.c*, and run it outside of the *edge* environment. Move the cursor to the console window and enter the commands:

```
cc  -g  sort.c  -o  sort
sort  -o  names.out  names.in
```

You have successfully debugged your program. Remember, if you want to take a break at this point, you can save your work on the code by typing:

```
make  save
```

Summary of Basic Commands

To give commands to *edge* you can type them at the prompt in the Command Window, select them from the command menu, or select them from the *edge* pop-up menu.

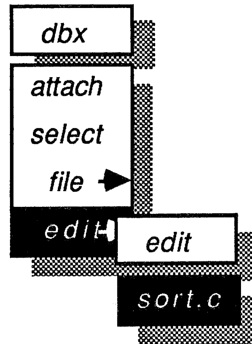
You learned three commands that you type in the Command Window. Square brackets ([]) surrounding an argument mean the argument is optional; angle brackets (<>) surrounding an argument mean it is mandatory.

- **edge** <*executable filename*>: Go into the *edge* environment.
- **run** [*arguments*]: Run the executable file with which you are currently working.
- **sh** <*command*>: Start up a new UNIX shell to execute this command.

You learned five commands that you select from the command menu.

Menu	Description
rerun	Rerun the last program using the same arguments.
cont	
step	
next	
where	Display details of the program fault.
interrupt	
sh	
quit	Exit from edge.
print	Display the value of the highlighted variable.
print *	
px	
stop at	Set breakpoint at highlighted line.
stop in	
cont to	
edit	
list	

You learned one command that you select from the pop-up menu.



Start up a UNIX shell that is running *vi* on this file.

You will use these commands extensively in the next chapter, along with several advanced commands, to help you track down more complex bugs.

2. More Elusive Bugs

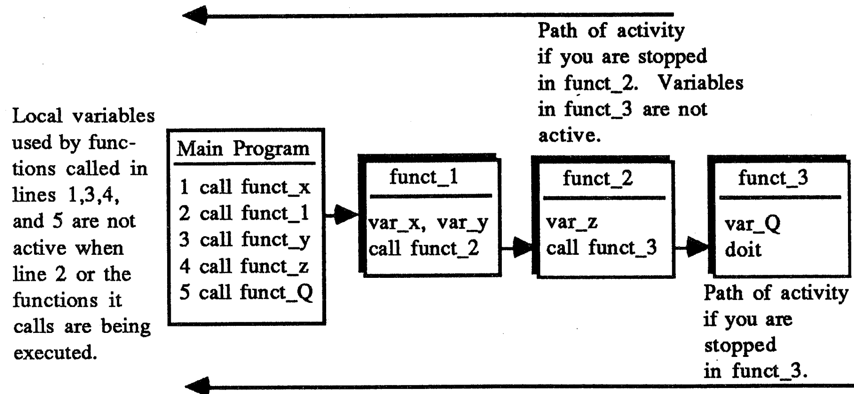
As you saw in Chapter 1, the basic *dbx* and *edge* commands are very useful and versatile. However, at times your programs will demand more sophisticated debugging tools. This chapter describes the advanced commands, and leads you through a more complex debugging situation.

Understanding Some Advanced Commands

You will learn to use 14 new commands in this chapter. As in Chapter 1, most of the commands are explained during the debugging session when you reach a point where you need to use them. However, some of the commands require more detailed explanations, so you will learn what they do now, and how to use them during the session.

The *trace* command lets you track the value of a variable as it changes. When you use *trace*, you must remember three important rules:

- You can trace only *active* variables. At any point during the execution of a program, the program has access to a certain set of variables; these variables are active at this point. Global variables are always active. Local variables are active only when their function either is being executed, or is calling a function that also has active variables. Such a series of functions calling other functions is called a *path of activity*. When you set a breakpoint using *edge*, the program stops at a certain point in its execution where there is a set path of activity. This path starts at the function in which you have stopped, and extends back through the intermediate functions to the line of the main program from which it all originated. Any variable along this path is active, and therefore you can trace it. (See the figure on the following page.)



- The syntax you use to give the *trace* command depends on your location within the path of activity. If you are stopped in the function *funct_3* and want to trace the variable *var_x* which is in *funct_1*, you must type `trace funct_1.var_x`. If you are already in *funct_1*, just type `trace var_x`.
- Always set a trace in the first executable line of code after the line that assigns the new value to the variable. This is necessary because *edge* displays the value of the variable before it executes the line at which you set the trace.

The *step* and *next* commands let you execute and view each line individually, effectively letting you step through your whole program.

step lets you go through your program in its logical order, one line at a time. When you get to a line that calls a function, the next line you will see is the first line of that function. When the function ends, you return to the line of code that called it.

next also lets you go through your program line by line, but it treats each line, even a line that calls a function, as a single event. So, when you reach a line that calls a function, the program executes it, but you don't step through the function code and watch it happen. Rather, you see the next line of code in the current function and you can check the values that the other function returns.

Both *step* and *next* display the line of code before it is executed. To check the value of a variable that is assigned on the current line, you must execute *step* or *next* one more time, and then *print* the variable.

Using the Advanced Commands

If you used the *make save* command to take a break from the tutorial, you can now pick up where you left off. You need to restore the files that you edited earlier, and then recompile *sort.* Return to the */usr/tutor/edge/C/src* directory and type:

```
make  restore
cc  -g  sort.c  -o  sort
```

Bug #3

1. Up to this point you have been working in the *src* directory. Since *sort* is working, make a copy of *sort*, place this copy in the *C* directory, and try it out there. Copy *sort* into *C*, and change directories so that *C* is your current directory.

   ```
   cp  sort  ..
   cd  ..
   ```

2. Sort the file *names.in*, and put the result into the file *names.out.* This time try using the *-i* flag so *sort* will ignore letter case.

   ```
   sort  -i  names.in  -o  names.out
   ```

3. You see this message:

   ```
   sort:  cant open input file (null)
   ```

   It seems that using *-i* caused a problem, so go into the *edge* environment.

   ```
   edge  sort
   ```

4. You notice that the Source Window didn't appear. This is because *edge* can't find your source code. *edge* assumes that source code and libraries for your program are in the current working directory unless you tell it otherwise. *sort.c* is still in the directory *src* while you are now in *C*. Tell *edge* which directories contain files that it needs to use.

```
use  src
```

5. Now that you can see your source code, search for the error message that you saw when you ran *sort*. *edge* supports the *vi* string search commands slash (/) and question mark (?). / searches forward through your file; ? searches backwards. Search forward for the first occurrence of *cant open*.

```
/cant open
```

6. The error message prints the value of a variable called *inputfile*. Use / to find the line of code in which *inputfile* is initialized. Press only / to find the second and third occurrences of the string.

```
/inputfile
/
/
```

7. You find that it is initialized in line 127. Set a breakpoint here by highlighting line 127, then selecting 'stop at' from the command menu.

8. Run *sort* in *edge* using the *-i* flag.

```
run  -i  names.in  -o  names.out
```

9. In the Command Window, you see this message:

```
Process  6155 (sort) started
Process  6155 (sort) finished
```

Because *sort* didn't stop, we know it never executed line 127.

10. Scroll through the code until you find the loop that processes the command line arguments. You find that the variable *curarg* keeps track of the value of the current argument.

11. Trace *curarg* as it changes values. Because *trace* displays the value of a line before it executes the line, be sure you set the trace at the first executable line after *curarg* is assigned a new value, line 86. Move the cursor to the Command Window and type:

    ```
    trace  curarg  at  86
    ```

12. Run the program again by selecting 'rerun' from the command menu.

    The cursor changes shape so it now looks like the corner of a window. *edge* displays the tracing information in a special Variable Display Window that you create (sweep out).

13. Sweep out the Variable Display Window:

    Position the cursor outside the *edge* windows. Press and hold the right mouse button to set the corner of the new window. While holding the right mouse button, drag the cursor diagonally to where you want the opposite corner to appear, then release the button.

    This is the Variable Display Window. You can scroll through this window just as you can scroll through the Source Window. The Variable Display Window displays the message:

    ```
    [2] curarg changed before [main: line 86]:

            new value = 0x7fffdbe5 = "-i";
    [2] curarg changed before [main: line 86]:
            old value = 0x7fffdbe5 = "-i";
            new value = 0x7fffdbf1 = "-o";
    ```

    curarg received some values, but didn't receive the value of the input file name. You will want to check out the dummy counter *i* which determines the value that *curarg* receives. But before you do this, find out which *edge* commands you have already set by using the *status* command.

14. Move the cursor to the Command Window and type:

```
status
```

The Command Window displays the list:

```
[2]  stop at "sort.c": 127
[3]  { ; trace curarg;} at "sort.c": 86
```

You should delete the *curarg* trace so it doesn't clutter the *i* trace. When you use the *delete* command, refer to the *edge* breakpoints and traces by using their status numbers.

15. Delete the trace command listed as status item three. Type:

```
delete  3
```

16. Now trace the dummy counter *i*. At the dbx prompt, enter the command:

```
trace  i  at  86
```

17. Run the program by selecting 'rerun' from the command menu. In the Variable Display Window, you see the message:

```
[3] i changed before [main: line 86]:
        new value = 1;
[3] i changed before [main: line 86]:
        old value = 1;
        new value = 3;
```

Notice that *i* skipped from 1 to 3. It seems that *i* is not being incremented properly. You can use *edge* to see what would happen if *i* received the value 2.

18. Set a breakpoint at the line in which the variable *i* receives its value, line 83. To set a breakpoint at line 83, highlight 83, then select 'stop at' from the command menu.

19. Rerun *sort* by selecting 'rerun' from the command menu.

20. Now you can tell *edge* that you want *i*'s value to be 2 by using the *assign* command.

```
assign i = 2
```

21. Continue running the program by selecting 'cont' from the command menu.

22. When *sort* stops at line 127, check to see if *i*'s new value changed *curarg*'s value. Highlight the word ''curarg'' in line 127, then select 'print' from the command menu.

    The value is *names.in*, as it should be. This shows that your program would work if *i* were incremented properly.

23. Continue running the program to make sure it works. Select 'cont' twice from the command menu. If you read the User Window, you see that the output file was named *names.out*.

24. Since *sort* faulted when you tried to use the ignore case option, scroll through the code that processes this option and check for places where *i* is incremented.

    You see that *i* is incremented twice in the case statement loop. It should be incremented only once.

25. Edit *sort.c* by placing the cursor in any *edge* window, pressing the right mouse button, and selecting 'src/sort.c' from the rollover menu that is beneath the 'edit' choice.

26. Tell *vi* to add line numbers.

```
:set number
```

27. Delete only these three lines:

```
95          /* bump the counter*/
96          i++;
97
```

Your code should now look like this:

```
93       ignorecase++;
94
95       break;
```

28. Save your changes and exit from *vi*.

```
:wq
```

29. Move the cursor over the Command Window and exit from *edge* by selecting 'quit' from the command menu.

Bug #4

1. Move the cursor to the console window.
2. Return to the *src* directory and recompile and run your program.

```
cd  src
cc  -g  sort.c  -o  sort
sort  -i  names.in  -o  names.out
```

3. Now make sure that it works without the *-i* flag.

```
sort  names.in  -o  names.out
```

4. The program seems to be working. Just to be positive, take a look at the output file.

```
more  names.out
```

5. As you can see, *sort* did not sort the list correctly. Go into *edge* to find the problem.

```
edge  sort
```

6. The C library routine *qsort* actually does the sorting, so look for the code that calls it.

```
/qsort
/
```

qsort depends on you to write a function that will compare records. (*qsort* is described in detail in the *IRIS-4D Programmer's Reference Manual*, section 3.)

7. Find your *compare_recs* function.

```
/compare_recs
```

When you find the function, you see that *compare_recs* uses another C library routine, *strcmp*, to perform the string comparison. (*strcmp* is also described in the *IRIS-4D Series Programmer's Reference Manual*, section 3, under the *string*(3c) routine.) Since these library routines should work, look at the input and output files to see if any type of sorting occurred. This may give you a clue about a possible bug in the implementation of these routines.

8. Compare *names.in* to *names.out* to see if any sorting occurred. To view the contents of a file other than the one you are debugging, use the *file* command. Type:

```
file  names.in
file  names.out
```

9. You see that the ordering did change, but rather randomly. Go back to your source file, and look at the *compare_recs* function again. To use *file* to view a source file in the Source Window, place the cursor in any *edge* window, press the right mouse button, and select the source file from the rollover menu that is beneath the 'file' choice. The 'file' choice lists all of the source files that are part of your program. In this case, select 'sort.c'.

10. List the function, *compare_recs*. Rather than search for the string *compare_recs*, you can use the *list* command. When you use *list* with a function name, *edge* takes you to the beginning of the function. At the dbx prompt, type:

```
list  compare_recs
```

11. Set a breakpoint at the end of this function so you can check the values of the variables before they are returned. Highlight line 239 and select 'stop at' from the command menu.

12. Run the program.

```
run  names.in  -o  names.out
```

The Command Window displays the message:

```
[2] Process 6194 (sort) stopped at [compare_recs:239, 0x400608]
              return(strcmp(rec0,rec1));
```

13. Check the values of the two variables *rec0* and *rec1*. First highlight *rec0* and select 'print', then highlight *rec1* and select 'print'.

    You see memory addresses.

14. Use the *whatis* command to find out what kinds of variables *rec0* and *rec1* are.

```
whatis  rec0
```

    The system tells us `unsigned char **rec0` and `unsigned char **rec1`. This indicates that *rec0* is a pointer to a string. *strcmp* needs strings, not pointers.

15. Find out to which string *rec0* points.

```
print  *rec0
```

    This is the first record in *names.in*. You need to change line 239 so that *strcmp* receives a string rather than a pointer.

16. Edit *sort.c* by placing the cursor in any *edge* window, pressing the right mouse button, and selecting 'sort.c' from the rollover menu that is beneath the 'edit' choice.

17. Add line numbers.

```
:set  number
```

18. Change line 239 so that it looks like this:

```
239            return(strcmp(*rec0,*rec1));
```

19. Save your changes and exit from *vi*.

```
:wq
```

20. Move the cursor to the Command Window and exit from *edge* by selecting 'quit' from the command menu.

Bug #5

1. Move the cursor to the console window.

2. Compile and run your program without the *-i* flag, and look at the output file.

```
cc  -g  sort.c  -o  sort
sort  names.in  -o  names.out
more  names.out
```

3. It seems to be working. Now try it with the *-i* flag.

```
sort  -i  names.in  -o  names.out
more  names.out
```

4. The comparison doesn't seem to work properly, so go into the *edge* environment.

```
edge  sort
```

5. Set a breakpoint in the *compare_recs* function. Use the *stop in* command. When you use *stop in* with a function, it sets a breakpoint at the first executable line of the function.

```
stop  in  compare_recs
```

6. Run the program in *edge*.

```
run  -i  names.in  -o  names.out
```

7. Go through *compare_recs* one step at a time. Use *step* in this case. Select 'step' from the command menu.

   In the Source Window, *edge* highlights the line:

   ```
   233           lower(tempbuf0,*rec0);
   ```

8. Select 'next' from the Command Window menu to execute the next line of code and see if *lower* is returning the correct value: the first record of the file *names.in*. Do not select 'step,' or you will descend into the function, *lower( )*.

9. Print the value of *tempbuf0* (in line 233) by highlighting it and selecting 'print' from the command menu. The displayed value is the string `mtsn`, which is not an element in the file to be sorted. This doesn't look correct.

10. Check the first element of the array *rec* to see what the record should be. Highlight **rec0* and select 'print'.

    The displayed value is `Smithson\n`. If you compare this to the contents of *tempbuf0*, it looks like *lower* is putting every other letter into the buffer.

11. Select 'step' to go into the function *lower*, and look at the code.

    Studying the code, you see that the variable *c* moves each letter of a record from the buffer *bufinput* into the buffer *result*. The contents of *result* are ultimately passed to the buffer *tempbuf*.

12. Check the above analysis of the code, trace *c*'s value at the end of the loop. Move the cursor to the Command Window and type:

    ```
    trace  c  at  257
    ```

13. Now check the contents of the buffer *bufinput* to see which record is about to be put into *result*. Highlight *bufinput* in line 251 and select 'print'.

14. Tell *edge* to continue execution. Select 'cont'.

15. Use the right mouse button to sweep out the Variable Display Window.

    For some reason *c* is moving only every other letter. There must be a problem where *c* is assigned a value.

16. Look at *lower* to find where c is assigned a value.

```
list  lower
```

17. It's probable that *iscap* is responsible, so list it.

```
list  iscap
```

18. If *iscap* is not defined, then it isn't a function. Use the *whatis* command to get some information about it.

```
whatis  iscap
```

19. Once again, it is not defined. Make sure *whatis* works by using it on *lower*.

```
whatis  lower
```

20. As expected, *whatis* works fine on the function *lower*. *whatis* can give you information about any variable, type, or function that is in your program. The only kind of structure *whatis* can't describe is a preprocessor directive, such as a macro, so *iscap* may be a macro.

21. Search the code for a macro definition of iscap.

```
/iscap
/
```

    You see that the macro *iscap* (defined on line 34) performs two substitutions. This is what your line of code looks like after *iscap* has been invoked.

```
if ( (((c=*bufinput++)>='A')&&((c=*bufinput++)<='Z')))
```

This expanded macro increments *bufinput* twice. You need to change your code so that *c* is assigned before *iscap* is invoked. This will prevent *bufinput* from being incremented by *iscap*.

22. Edit *sort.c* by placing the cursor in any *edge* window, pressing the right mouse button, and selecting 'sort.c' from the rollover menu that is beneath the 'edit' choice.

23. Add line numbers.

```
:set number
```

24. Change lines 251 and 252 so they look like this:

```
251         while (c = *bufinput++) {
252                 if (iscap(c))
```

25. Save your edits and exit from *vi*.

```
:wq
```

26. Move the cursor over the Command Window and exit from *edge* by selecting 'quit' from the command menu.

27. Move the cursor over the console window then recompile your program, run it, and check the results.

```
cc  -g  sort.c  -o  sort
sort  -i  names.in  -o  names.out
more  names.out
```

You have completely debugged your program, and you are through using this directory. Before you go on to the last chapter, restore the */usr/tutor/edge/C/src* directory to its original form so that other people can use it. To do this, type:

```
make  done
```

Summary of Advanced Commands

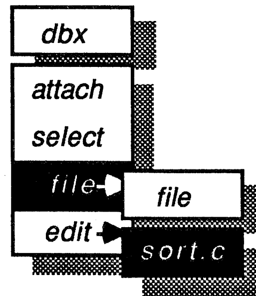
You learned eight commands that you type in the Command Window. Square brackets ([]) surrounding an argument mean the argument is optional; angle brackets (<>) surrounding an argument mean it is mandatory.

- **use** *<directory>* [*directory*] ... : Use these directories. They contain source code or the libraries that the program uses.
- **file** *<filename>*: Make this file the current file and display it in the Source Window. Type this command in the Command Window when the file you want to display is not a source file.
- **status**: Show a list of all of the *edge* breakpoints and traces that are currently set.
- **delete** *<status number>* [*status number*]: Delete this command.
- **trace** *<variable>* **at** *<line number>*: Print the value that this variable has when it reaches this line number.
- **stop in** *<function>*: Stop the program when it enters this function, and print the first executable line.
- **assign** *<variable>* = *<value>*: Assign a certain value to a variable.
- **whatis** *<object>*: Display the definition of this object (function, type, or variable).

You learned three additional commands from the command menu.

Menu	
rerun	
cont	Continue execution of a stopped program.
step	Execute next line of code. Step down into functions.
next	Execute next line of code. Do not step down into functions.
where	
interrupt	
sh	
quit	

You learned one command that you select from the pop-up menu.



Display this file in the Source Window and make it the current file.

You also learned these *vi* search commands:

- /<*string*>: Search forward through the file for this string.
- ?<*string*>: Search backward through the file for this string.

This list and the list of basic commands on pages 13 and 14 cover most of the *edge* commands you need to debug your programs. A complete list of all *edge* commands that you learned in this tutorial appears in Chapter 3.

3. On Your Own

At this point you know enough about *edge* to use it to debug your own non-graphics programs. The first section of this chapter gives you some information on debugging graphics programs using *edge*. The rest of the chapter provides three useful references: a table that summarizes the debugging process, a list of all *edge* commands that you learned in this tutorial, and a list of sources that contain additional information about *edge*.

Using *edge* to Debug Graphics Programs

You can use all of the *edge* commands that you learned in this tutorial to debug graphics programs. The one difference is that you must run graphics programs in the foreground when you run them under *edge*. This section describes two ways you can do this.

To use the first method you must call the `foreground` routine in your source code. At the beginning of the *main* function, add this line:

```
foreground();
```

To use the second method you must add a conditional statement to your code so that when you use the *-D* flag when you compile, the compiler adds the `foreground` call to your code. This way the call happens only when you need it. At the beginning of your *main* function, add this code:

```
# ifdef DEBUG
                foreground();
# endif
```

If your program were called *graphic.c* and you wanted to debug it, you would compile it by typing:

```
cc  -g  -DDEBUG  graphic.c  -o  graphic  -Zg
```

The Debugging Process

This table illustrates a good, general purpose procedure for systematically debugging your own programs. Commands that you type at a prompt are printed here in `typewriter font`.

Procedure	*edge* Commands
1. Compile your program using the debugging flag.	`cc -g`

2. Run your newly compiled program in the *edge* environment. Tell *edge* which directories it must use.	`edge <filename>`
	`use <dir> [dir]`
	`run [arguments]`

3. If the program does not fault, go to step #4. If it does fault, find where the fault occurred.	select "where"
4. Look over the code and set break-points at various lines and functions to check values.	highlight the code and select "stop" `stop in <function>`
5. Rerun your program with the same arguments.	select "rerun"
6. When the program stops at each breakpoint, look at values,step through the code if necessary, and continue running the program.	highlight a variable and select "print" select "stop" select "next" select "cont"
7. If the value of a variable is not correct, trace it at the line after it is assigned its value. Remember to specify its module and function if necessary.	`trace [mod].[funct].<var> at <line number>`
8. Keep track of breakpoints and traces and delete those that you no longer need.	`status` `delete <status number>`
9. When you find the bug, edit your code.	select a file from the "edit" sub-menu
10. Exit from *edge*and go back to step #1.	select "quit"

Summary of *edge* Commands

This section contains all of the *edge* commands that you can issue by typing in the Command Window, selecting from the command menu, or selecting from the pop-up menu.

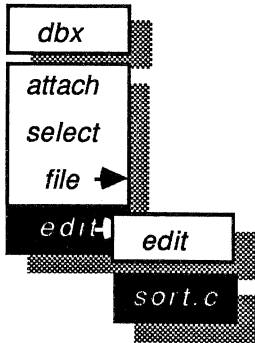
Textual Commands

- **assign** *<variable>=<value>*: Assign a certain value to a variable.
- **delete** *<status number>* [*status number*]: Delete the commands that have these status numbers.
- **edge** *<executable filename>*: Go into the *edge* environment.
- **file** *<filename>*: Make this file the current file.
- **list** [*function*]: Display the code for this function.
- **run** [*arguments*]: Run the executable file with which you are working.
- **status**: Show a list of all the *edge* breakpoints and traces that are currently set.
- **stop in** *<function>*: Stop the program when it enters this function, and print the first executable line.
- **trace** *<variable>* **at** *<line number>*: Print the value that this variable has when it reaches this line number.
- **use** *<directory>* [*directory*] ... : Use these directories. They contain source code or libraries that the program uses.
- **whatis** *<object>*: Display the definition of this object (function, type, or variable).

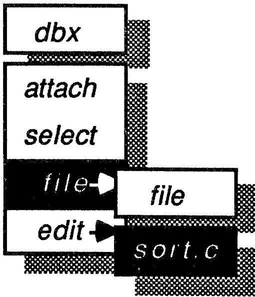
Choices on the Command Menu

Command	Description
addproc	Add highlighted process to pool of processes controlled by edge.
delproc	Delete highlighted process from pool of edge-controlled processes.
suspend	Suspend execution of highlighted process.
activate	Select process from pool of processes controlled by debugger.
debug	Add selected process to process pool and stop process.
rerun	Rerun the last program using the same arguments.
cont	Continue execution of a stopped program.
step	Execute next line of code. Step down into functions.
next	Execute next line of code. Do not step down into functions.
where	Display details of the program fault.
interrupt	Stop edge from completing the current command.
sh	Start a new UNIX shell.
quit	Exit from edge.
print	Display the value of the highlighted variable.
print *	Display the value pointed to by the highlighted variable.
px	Display the hexedecimal value of the highlighted variable.
stop at	Set breakpoint at highlighted line.
stop in	Set break point at start of function containing highlight.
cont to	Continue execution of program until the highlighted line.
edit	Edit source for highlighted function.
list	List source for highlighted function.

Choices on the Pop-up Menu



Start up a UNIX shell that is running *vi* on this file.



Display this file in the Source Window and make it the current file.

vi Search Commands

- / <*string*>: Search forward through the file for this string.
- ? <*string*>: Search backward through the file for this string.

Where to Find Additional Information

The *IRIS-4D Programmer's Reference Manual*, section 1, contains two relevant manual pages: *edge*(1) describes all of the *edge* commands and command line options; *dbx*(1) describes all of the *dbx* commands and command line options. The same manual pages are on-line. To view them, type:

```
man  edge
```

or

```
man  dbx
```