Learning to Debug with edge

FORTRAN Edition
Learning to Debug with *edge*

FORTRAN Edition

*Version 2.0*
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
FORTRAN Edition
Version 2.0
Document Number 007-0904-020

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 ........................................... 11
   Summary of Basic Commands ....................... 14

2. More Elusive Bugs ............................... 17
   Understanding Some Advanced Commands .......... 17
   Using the Advanced Commands .................... 19
   Bug #3 ........................................... 19
   Bug #4 ........................................... 25
   Summary of Advanced Commands ................. 30

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 FORTRAN 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 to have a very 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.

Source File

Create your source code as usual.

Compile

Compile using the debugging flag (-g).

Object File

symbol table

The compiler creates an expanded symbol table in your object file.

Link

Your file is linked with the debugging flag.

Executable File

Your executable file contains the information that the debugger needs.

2 Learning to Debug with edge, FORTRAN Edition

IRIS-4D Series
Using Makefile to Set Up the edge Tutorial

You will be working on a sample program called sort.f. During this session you use six 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/fortran/src. Type:

cd /usr/tutor/edge/fortran/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.f, sort.h, and sort.m. The program sort.f reads the input file names.in, sorts it, and puts the results into an output file. To briefly look over sort.f, type:

more sort.f

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.f, 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/edgefortran/src` directory and type:

```
make restore
```

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

```
make done
```

Now you are ready to tackle the first bug.
Bug #1

1. Compile and link sort.f using the edge flag, and name your executable file sort.
   
   ```
f77 -g sort.f -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
   ```

3. You see this message:

   ```
sort: cant open input file named
   ```

   `sort` couldn't open `names.in`, and also couldn't report its name. You want to use `edge` to find the problem, so go into the `edge` environment.

   ```
edge sort
   ```

4. You see the three `edge` windows. They are described 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 commands 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 bar 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 Source Window also continues the command menu started in the Command Window.

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.

5. Look for the section of code where the input file is assigned. Scroll to line 122.
6. Set a breakpoint at line 122 to make the program stop and display this line when it reaches it. To set a breakpoint at a line of code, highlight the line of code, then select ‘stop at’ from the command menu.
7. Now run sort in edge. Type:

    run names.in -o names.out

In the Command Window you see this message:

```
Process 7995 (sort) started
[2] Process 7995 (sort) stopped at [MAIN:122 , 0x4003bc]
    inputfile = curarg
```

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

8. 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 at the beginning of the word, pressing and holding the left mouse button, then 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.

   The value of curarg is """, a null string. Although it is possible that curarg is supposed to be empty, it is also possible that is was never initialized. Therefore, you should check to see whether it was initialized.

9. Scroll back to where curarg should have been initialized, between lines 89 and 92. As you can see, curarg has not been initialized.
10. Edit the source file. To edit your file, position your cursor in any edge window, and press and hold the right mouse button. You see the following menu:

```
  dbx
    attach
    select
    file ➡
    edit ➡
```

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.f’. Make sure it is highlighted (your cursor should be on top of it), then release the mouse button.

```
  dbx
    attach
    select
    file ➡
    edit
    sort.f
```

You see a red outline, and the shape of your cursor has changed. 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.f. (When your program consists of several source files, the ‘edit’ sub-menu contains all of them so you can access them easily.)

11. Tell vi to display line numbers. Move the cursor to vi window and type:

   :set number

This step is very important for maintaining the integrity of this tutorial. You must add temporary line numbers to your file so that you can edit it exactly as this tutorial does. This way, the references to line numbers throughout the tutorial will remain accurate.
12. Edit the code so that lines 87-95 look like this:

```fortran
87  argc = iargc()
88  do 100 i=1, argc
89  c get the current argument
90  call getarg(i, curarg)
91  c is it a switch?
92  
93  
94  
95  
```

13. Save your edits and exit from vi as usual. Type:

```
:wq
```

14. When you exit vi, the new shell disappears.

15. Exit from edge by selecting ‘quit’ from the command menu.

You have successfully eliminated the first bug.
Bug #2

1. Recompile your program using the edge flag, then run it.

    f77 -g sort.f -o sort
    sort -o names.out names.in

2. You see this 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. Go into the edge environment.

    edge sort

3. 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 using the middle mouse button.

4. Set a breakpoint at the line in which the name of the output file is assigned. Highlight the line of code (line 112) using the left mouse button, then select 'stop' from the command menu.

5. Use the run command to run sort in edge.

    run -o names.out names.in
6. You see that line 112 contains the variable \texttt{curarg}. Check \texttt{curarg}'s value by highlighting it using the left mouse button, then selecting 'print' from the command menu.

The value is \texttt{-o}. This is the argument that appears on the command line one position before the desired output file, \texttt{names.out}. This means that the dummy counter \texttt{i} has not been incremented properly. If you scroll through this loop of code, see that you need to increment \texttt{i} past the \texttt{-o} switch.

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

8. Tell \texttt{vi} to display line numbers.

\begin{verbatim}
:set number
\end{verbatim}

9. Change your code so that lines 106-112 look like this:

\begin{verbatim}
106     elseif (curarg(2:2) .eq. 'o') then
107         c     the output file name follows
108
109         c     increment past the switch
110     call bump(i)
111
112         c     get the output file name
\end{verbatim}

10. Save your changes and exit from \texttt{vi}:

\begin{verbatim}
:wq
\end{verbatim}

11. Exit from \texttt{edge} by selecting 'quit' from the command menu.

12. Recompile \texttt{sort.f}, and run it outside of the \texttt{edge} environment. Move the cursor to the console window and type:

\begin{verbatim}
    f77 -g sort.f -o sort
    sort -o names.out names.in
\end{verbatim}
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 two commands that you type. 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.

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

```
rerun
cont
step
next
where
interrupt
sh
quit

print
print *
px
stop at
stop in
cont to
edit
list
```

Exit from edge.

Display the value of the highlighted variable.

Set breakpoint at highlighted line.
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 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 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 routine either is being executed, or is calling a routine that also has active variables. Such a series of routines calling other routines 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 routine in which you have stopped, and extends back through the intermediate routines 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 routine `rout_3` and want to trace the variable `var_x` which is in `rout_1`, you must type `trace rout_1.var_x`. If you are already in `rout_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 routine, the next line you will see is the first line of that routine. When the routine 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 routine, as a single event. So, when you reach a line that calls a routine, the program executes it, but you don’t step through the routine code and watch it happen. Rather, you see the next line of code in the current routine and you can check the values that the other routine returns.

Both *step* and *next* display the line of code before it is executed. To check the value of a variable that is in the current line, 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/fortran/src` directory and type:

```
make restore
f77 -g sort.f -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 *fortran* directory, and try it out there. Copy *sort* into *fortran*, and change directories so that *fortran* 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
   ```

You see this message:

```
sort: can't open input file named
```

Version 2.0

More Elusive Bugs 19
3. It seems that using -i caused a problem, so go into the edge environment.

   edge sort

   You notice that the Source Window did not 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.f is still in the directory src while you are now in fortran.

4. 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 receives a variable called inputfile. Use / to find the line of code in which inputfile is initialized.

   /inputfile
   /
   /

7. You find that inputfile is initialized in line 128. Set a breakpoint here by highlighting line 128, then selecting ‘stop at’ from the command menu.

8. Run the program in edge using the same flags as before.

   run -i names.in -o names.out
9. In the User Window you see this message:

```
sort: can't open input file named
```

This shows you that something else is wrong. `sort` executed completely, but didn’t stop at line 128. This means that it never looked at 128. Look for the loop of code that processes the command line arguments by using the middle mouse button to scroll through your source code.

You find that the variable `curarg` keeps track of the value of the current argument (either a flag or a file name). Trace `curarg` as it changes values. Be sure to set the trace at the first executable line after `curarg` is assigned a new value, since `trace` displays the value of a line before it executes the line. At the `(dbx)` prompt, type:

```
trace curarg at 96
```

10. Run the program again. When 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. The cursor changes shape so it now looks like the corner of a window. `edge` displays the tracing information in a special window that you create (sweep out).
11. To sweep out the Variable Display Window, position the cursor above all of the edge windows, press and hold the right mouse button to set the corner of the new window, 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. In the Variable Display Window, you see this message:

```
[2] curarg changed before [MAIN: line 5]:
    new value = "-i"
[2] curarg changed before [MAIN: line 96]:
    old value = "-i"
    new value = "-o"
```

curarg received some values, but didn’t receive the value of the input file name. Check out the dummy counter i which determines the value that curarg receives. Before you do this, find out which edge commands you have already set by using the status command.

status

12. You see this list:

```
[2] stop at "sort.f": 128
[3] { ; trace curarg; } at "sort.f" 96
```

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.

delete 3
13. Now trace the dummy counter $i$.

`trace i at 96`

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

```plaintext
[3] sort.MAIN.i changed before [MAIN: line 96]:
   new value = 1;
[3] sort.MAIN.i changed before [MAIN: line 96]:
   old value = 1;
   new value = 3;
```

Notice that $i$ skipped from 1 to 3. It seems that $i$ is not being incremented properly. Since `sort` didn't work correctly when you used the `-i` flag, scroll to the code that passes the dummy counter through the `-i` case.

You see that $i$ is incremented once at the beginning of the loop, and again at the end of the loop. Usually you increment a dummy counter in the `do` statement at the beginning of the loop. Edit `sort.f` by placing the cursor in any `edge` window, pressing the right mouse button, and selecting `src/sort.f` from the rollover menu that is beneath the 'edit' choice.

16. Tell `vi` to add line numbers.

`:set number`
17. Delete only these three lines.

103 c     bump the counter
104     call bump(i)
105

Your code should now look like this:

100 c     if (curarg(2:2) .eq. 'i') then
101         ignorecase = 1
102
103     elseif (curarg(2:2) .eq. 'o') then
104     the output name follows

18. Save your changes and exit from vi.

:wq

19. Exit from edge by selecting 'quit' from the command menu.
Bug #4

1. Return to the src directory and recompile and run your program.

   cd src
   f77 -g sort.f -o sort
   sort -i names.in -o names.out

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

   more names.out

3. The comparison doesn’t seem to work properly. sort is not ignoring the case of the records. Use edge to find the problem.

   edge sort

4. Look over the file names.in to make sure nothing has happened to it. To view the contents of a file other than the one you are debugging, use the file command.

   file names.in

5. It seems to be intact, so return your source file to the Source Window. 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.f’.

6. The routine bsort actually does the sorting, so find this routine.

   /bsort
   /

Version 2.0

More Elusive Bugs  25
7. You find that bsort uses the results of the routine cmprec. Find this routine.

    /cmprec
    /
    /
    /

8. Set a breakpoint in cmprec using the stop in command. When you use stop in with a routine, it sets a breakpoint at the first executable line of the routine.

    stop in cmprec

9. Run the program in edge.

    run -i names.in -o names.out

10. You see this message:

    Process 8350 (sort) started
    [2]Process 8350 (sort) stopped at [sort.cmprec:231 ,0x40080c]
        if(ignorecase .ne. 0) then

    Go through cmprec one step at a time. Select ‘step’ from the command menu.

11. You see this message:

    Process 8350 (sort) stopped at [sort.cmprec:235 ,0x400820]
        tempbuf0 = lower(rec(index0))

    If you take another step, you will enter the subroutine lower. Rather than stepping through it, use next to skip the explicit tracing of lower. Select ‘next’ from the command menu.
12. Check to see if tempbuf0 contains the right value, that is, the first record of the file names.in. Highlight tempbuf0 then select 'print' from the command menu.

13. This doesn’t look correct. Check the first element of the array rec to see what the record should have been. To do this you must be in the routine lower, so rerun sort and step through cmprec into lower. Select 'rerun' from the command menu, then select 'step' twice from the command menu.

14. Now check the value of rec(index0). Normally you could highlight this variable then select 'print' from the command menu. However, dbx doesn't recognize parentheses, so you need to use square brackets instead. Type:

    print rec[index0]

15. If you compare this to the contents of tempbuf0, it looks like lower is lowercasing only the first letter and putting it into the buffer. You see in the Source Window that the variable c moves each letter of a record from the buffer bufinput into the buffer result. isloweer checks the case of each letter, and tolower lowercases any upper case letters that isloweer finds. Trace c's progress by checking its value at the end of the loop.

    trace c at 279

16. Now check the contents of the buffer bufinput to see which record is about to be put into tempbuf. Highlight bufinput in line 266 and select 'print' from the command menu.

17. Tell edge to continue tracing by selecting 'cont' from the command menu.

18. Use the right mouse button to sweep out the Variable Display Window. The new value is an empty string. It seems that one of the subroutines is returning unprintable characters. Check out the routine tolower. Rather than search for the string tolower, you can use the list command. When you use list with a routine name, edge takes you to the beginning of the routine.

    list tolower
19. If `tolower` is not defined, then maybe it isn’t a routine after all. Use the `whatis` command to get some information about it.

```
whatis tolower
```

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

```
whatis lower
```

21. `whatis` can give you information about any variable, type, or routine that is in your program. The only kind of structure it can’t describe is a preprocessor directive, such as a macro; so, `tolower` may be a macro. Find `tolower` and check it out.

```
/tolower
/
```

22. `tolower` is indeed a macro, and it looks correct. It expects a capital letter from `islower` and lowercases it. Perhaps `islower` is passing something other than only capital letters. Find `islower`.

```
/islower
/
```

23. You see that there is a mistake in the macro `islower`. The z should be a capital letter. Edit `sort.f` by placing the cursor in any `edge` window, pressing the right mouse button, and selecting `sort.f` from the rollover menu that is beneath the `edit` choice.

24. Add line numbers:

```
:set number
```

25. Change line 25 so that it looks like this:

```
25   #define islower(c) ((c \geq \text{'A'}) \&\& (c \leq \text{'Z'})
```

26. Save your edits and exit from `vi`.

```
:wq
```

28 Learning to Debug with `edge`, FORTRAN Edition
27. Exit from *edge* by selecting ‘quit’ from the command menu.

28. Recompile your program, run it, and check the results. Move the cursor
to the console window and type:

```
    f77 -g sort.f -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/fortran/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 seven 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.
- **whatis <object>** : Display the definition of this object (function, type, or variable).

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

| addproc | Rerun the last program using the same arguments. |
| delproc | Continue execution of a stopped program. |
| suspend | Execute next line of code. Step down into functions. |
| activate | Execute next line of code. Do not step down into functions. |
| debug   |                                             |
| rerun   |                                             |
| cont    |                                             |
| step    |                                             |
| next    |                                             |
| where   |                                             |
| interrupt |                                             |
| sh      |                                             |
| quit    |                                             |
You learned one command that you select from the pop-up menu.

```
+-------------------+
| dbx               |
| attach            |
+-------------------+ select
+-------------------+     file
| edit              |
+-------------------+     sort.f
```

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 routine, add this line:

```
call 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 routine, add this code:

```
#if define DEBUG
   call foreground
#endif
```

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

```
f77 -g -DDDEBUG graphic.f -o graphic -g
```
# 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.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>edge Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Compile your program using the debugging flag.</td>
<td><code>f77 -g</code></td>
</tr>
<tr>
<td>2. Run your newly compiled program in the <code>edge</code> environment. Tell <code>edge</code> which directories to use.</td>
<td><code>edge &lt;filename&gt;</code> &lt;br&gt; <code>use &lt;dir&gt; [dir]</code> &lt;br&gt; <code>run [arguments]</code></td>
</tr>
<tr>
<td>3. If the program does not fault, go to step #4. If it does fault, find where the fault occurred.</td>
<td><code>select 'where'</code> &lt;br&gt; <code>highlight the code and select 'stop'</code> &lt;br&gt; <code>stop in &lt;routine&gt;</code></td>
</tr>
<tr>
<td>4. Look over the code and set breakpoints at various lines and routines to check values.</td>
<td><code>select 'rerun'</code> &lt;br&gt; <code>highlight a variable and select 'print'</code> &lt;br&gt; <code>select 'step'</code> &lt;br&gt; <code>select 'next'</code> &lt;br&gt; <code>select 'cont'</code> &lt;br&gt; <code>trace [mod].[rout].&lt;var&gt; at &lt;line number&gt;</code></td>
</tr>
<tr>
<td>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 routine if necessary.</td>
<td><code>status</code> &lt;br&gt; <code>delete &lt;status #&gt; [status #]</code> &lt;br&gt; <code>select a file from the 'edit' sub-menu</code> &lt;br&gt; <code>select 'quit'</code></td>
</tr>
<tr>
<td>8. Keep track of breakpoints and traces and delete those that you no longer need.</td>
<td></td>
</tr>
<tr>
<td>9. When you find a bug, edit the code.</td>
<td></td>
</tr>
<tr>
<td>10. Exit from <code>edge</code> and go back to step #1.</td>
<td></td>
</tr>
</tbody>
</table>
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**

- **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

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