Fx Debugger User Guide

absoft
development tools and languages
2781 Bond Street
Rochester Hills, MI 48309
U.S.A.
Tel: (248) 853-0095
Fax: (248) 853-0108

Chapter One

Introduction

When the state of the art user interface consisted of a card reader and line printer and the majority of programs where written in assembly language for mainframe computers, debugging programs involved long hours examining source listings and core dumps. Other debugging techniques included inserting code to print the values of variables and trace the flow of program execution, staring off into space, and blaming the hardware. As both hardware and software technology progressed, the need for more sophisticated debugging tools led to the development of interactive assembly and source level debuggers.

About Fx

Although interactive debuggers have existed for a number of years, a surprising number of programmers do not use them during program development. Many may have tried to use one and found the experience so frustrating that they returned to more primitive debugging techniques. Others, particularly FORTRAN and assembly language programmers, may have found that available debuggers fail to provide adequate support for their language of choice.

Absoft has created Fx to meet the needs of both the novice and experienced programmer. Fx provides both standard debugging capabilities, such as breakpoints and variable display, and includes advanced features like variable monitors and dynamic windows. FORTRAN programmers will find that Fx supports their language, and assembly language users will appreciate the support provided for machine language debugging.

Some people enjoy using a point and click graphical interface; others may find it a distraction, or not have access to the necessary hardware. Fx provides interfaces to suit both of these needs. When a graphics terminal is available, Fx allows you to take advantage of a graphic interface and still retain the convenience of a command line. When you are debugging on a character based terminal, Fx provides a command line interface and provides windowing capabilities where possible.

About Fx Interfaces

This manual has been structured to allow you to begin using the Fx graphical interface as quickly as possible. The interface is based on the Open Software Foundation's OSF/Motif™ user environment. You can begin debugging a simple program by entering “xfx” and pressing the Return key. If you wish to use the Fx character interface, you should refer to Appendix A. This appendix contains information on starting and stopping Fx when using the character interface, as well as details on interacting with Fx when using this interface.

About the Examples in this Manual

All Fx interfaces support command line entry. You should keep in mind that there might be more convenient ways to specify a given command. Although the examples in this manual use the complete spelling of each command, all commands can be entered by specifying the minimum number of characters required to distinguish the command. For example, the **quit** command can be abbreviated to **q**. When two commands begin with the same sequence of characters, the one which is more commonly used will be executed. For example, the **step** command can be abbreviated to **s**, while the **search** command can be abbreviated to **sea**.

Conventions used in this Manual

There are a few typographic and syntactic conventions used throughout this manual for clarity.

- The names of Fx menu items, commands and control variables, as well as Fx and compiler options, are shown in **bold**.
- Examples and sample command lines are shown in `Courier`.
- `{}` braces indicate that a syntactic item is optional.
- ... indicates a repetition of a syntactic element.
- *Italics* are used for emphasis and argument substitution.

An Overview of the Chapters in this Manual

- **Chapter Two - Tutorial**

  Describes how to start a debugging session, load your program for source level debugging, set breakpoints and quit the debugger.

- **Chapter Three - Fx Interface Reference**

  Presents all the features of the Fx graphical interface, including menus, commands, buttons and windows.

- **Chapter Four - Running Fx**

  Describes how to prepare your program for debugging, how to use core files, different methods of examining source code, and recreating a debugging session.

- **Chapter Five - Executing Programs in Fx**

  Introduces information on executing programs within the debugger, including using breakpoints, animating program execution and calling program procedures.

- **Chapter Six - Working with Variables**

  Discusses how to display variables and arrays, modifying variables, and finding overwritten variables.

- **Chapter Seven - Assembly Language Debugging**

  Provides details on using Fx to debug at the assembly language level. Topics include how to examine disassembled code, execute single instructions, and display the contents of registers and memory.

- **Chapter Eight - Command Arguments**

  Describes how to enter symbol names and constants as arguments to Fx commands. Included are a list of supported data types and operators for both C and FORTRAN, and the rules used when evaluating expressions.

- **Chapter Nine - Command Reference**

  Contains information on all Fx commands including each command’s purpose, arguments, and abbreviation. Notes and examples are provided where needed.

- **Appendix A - The Fx Character Interface**

  Introduces the Fx character interface, describes how to begin a debugging session with the graphic interface, and how to manipulate character based windows and interpretation of special keys.

- **Appendix B - Fx Control Variables**

  Presents a table of all the Fx control variables. Information on each variable includes its name, purpose, default value, and the values that it may be assigned.

- **Appendix C - Debugging with Optimized Code**

  Provides details on using the Fx debugger when working with optimized code.

Chapter Two

Tutorial

If you have never used a source level debugger before, you may be surprised by how useful a debugger can be in isolating problems in a program. This tutorial introduces the basics of Fx and provides step-by-step instructions that cover the following topics:

- **Launching the Debugger**

  Describes how to start a debugging session and introduces the basic features of the Fx Debugger interface.

- **Setting Breakpoints**

  Provides an introduction to using breakpoints to analyze sections of code.

- **Executing Programs**

  Describes how to run a program under Fx and introduces different menu selections for working with variables while executing the program.

- **Quitting the Debugger**

  Describes how to exit from Fx.

Launching the Debugger

This section discusses how to start Fx and load a program for a debugging session. It describes how to specify the name of the program to debug, the name of a core file, and the directories that contain the source code for the program.

The Tutorial Program

The tutorial program, `wrdcnt`, counts the number of words, lines, and characters in a text file. It is used to demonstrate the basic features of the Fx Debugger.

Before launching Fx, the executable version of the tutorial program must be created in the shell using the `make` command. The tutorial source file and a makefile that builds the tutorial program are included in the Fx Help Library. To get a copy of these files and create the tutorial program, execute the following:

To...	***Do...***

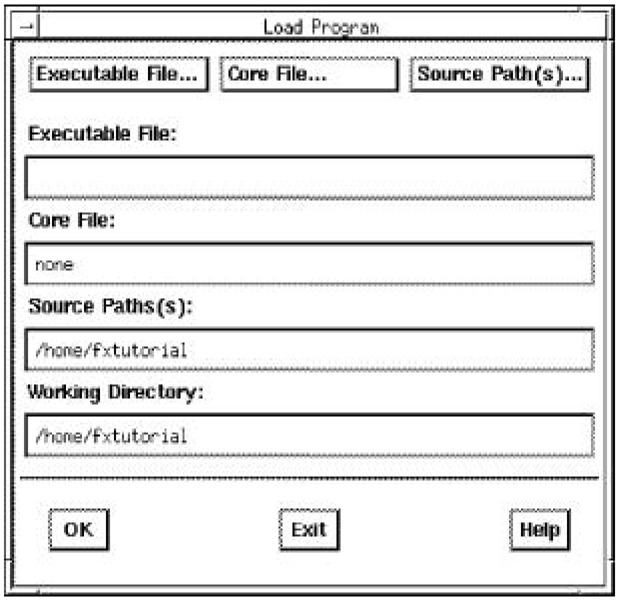
Get a copy of the source file and the makefile.	At the UNIX shell prompt, enter `ar -xv /usr/lib/Fxhelp.a wrdcnt.c Makefile` and press the Return key.
Create the tutorial program.	Type `make` at the shell prompt.

After compiling the tutorial program, the Fx Debugger is launched from the shell prompt to begin a debugging session:

To...	***Do...***
Launch the debugger.	Type `xfx` at the UNIX shell prompt and press the Return key.

Using the Load Program Dialog

When the debugger is launched, the **Load Program** dialog will appear. At the bottom of the dialog a **Help** button is available: click on this button to see information on the features of the dialog. If, for some reason, you need to quit from the debugger at this point, click the **Exit** button.



Executable File:

Core File: none

Source Paths(s): /home/fxtutorial

Working Directory: /home/fxtutorial

Figure 2-1
Load Program dialog

The four text fields: **Executable File**, **Core File**, **Source Path(s)**, and **Working Directory**, allow you to type in the names of the files and directories the debugger will use.

Executable File

This text field names the program to be debugged. The name of tutorial program, `wrdcnt`, will be entered into this text field.

Core File

This text field allows you to specify a core file. When an operating system creates a core file of a program, Fx can use this file to easily determine the location in your code where an error occurred. For the purpose of this tutorial, a core file will not be used.

Source Path(s)

This text field lists the directories that contain the source files that were compiled to produce the program you are debugging.

Working Directory

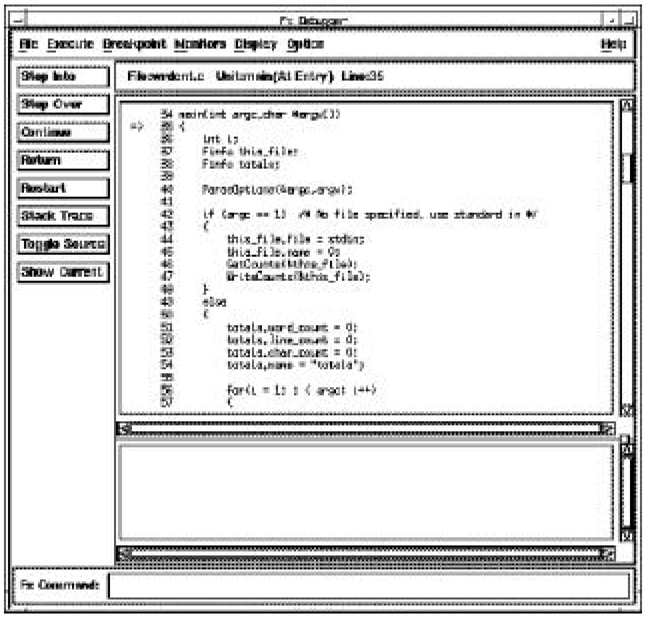
This field allows you to specify the directory that will be your program's current directory while you are debugging with Fx.

Loading the Program Into the Debugger

The tutorial program is loaded into the debugger by using the four text fields in the **Load Program** dialog. Since Fx has supplied suitable defaults for the source paths and working directory fields, you only need to enter the name of the executable file.

To...	***Do...***
Specify the executable file.	Type `wrdcnt` into the **Executable File** text field.
Load the program for debugging.	Click the **OK** button.

After the program is loaded, the source code is displayed in the main window, the Fx Debugger window. This window is where you will spend much of your time, executing commands necessary to debug your program.



```
File:wordcnt.c  Unit:main(At Entry)  Line:35
```

```
    main(int argc,char *argv[])
=>  {
        int i;
        FInfo this_file;
        FInfo totals;

        ParseOptions(&argc,argv);

        if (argc == 1)  /* No file specified, use standard in */
        {
            this_file.file = stdin;
            this_file.name = 0;
            GetCounts(&this_file);
            WriteCounts(&this_file);
        }
        else
        {
            totals.word_count = 0;
            totals.line_count = 0;
            totals.char_count = 0;
            totals.name = "totals";

            for(i = 1; i < argc; i++)
            {
```

Figure 2-2
Fx Debugger window

The **Status Field** is located under the **Menu Bar**; it shows where you are in your program, the name of the source file, the program unit, and the line number.

Underneath the **Status Field** is the **Source Pane**, which shows the program's source code. By pressing the arrows on the scroll bar, different sections of the source code can be viewed. The **Output Pane** is located under the **Source Pane** and displays the output from Fx commands (like printing the values of variables). It also has a scroll bar to view output. This pane can be resized by using the small square, called a **sash**, at the top of the scroll bar.

The **Command Entry Field** is available for invoking Fx commands without using the menus. To invoke a command from this field, type in the command and press the Return key. Located along the side of the Fx Debugger window are push buttons that invoke Fx commands directly from the window.

Setting a Breakpoint

Breakpoints allow you to suspend execution of your program at a specific location. You can set breakpoints on source lines, procedures and even individual machine instructions. Before we execute the program, we need to set two breakpoints: one at a source line and another inside a specific procedure.

Setting a Breakpoint on a Line

If you triple-click with the first mouse button on a line of source code or within a line, and then press the third mouse button, the **Source Pane** popup menu will appear. This menu contains items for setting breakpoints, printing variables and viewing procedure code.

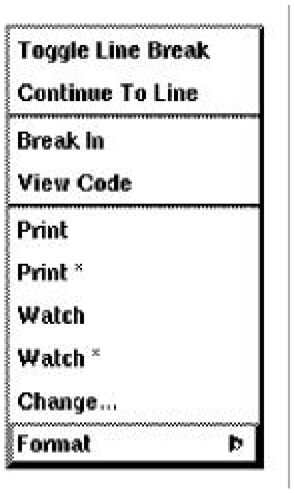


Figure 2-3
Source Pane popup menu

To select a menu item from this menu, hold down the third mouse button, drag the cursor to the item, and then release the button. We will use the **Toggle Line Break** item to set a breakpoint at line 40.

To...	*Do...*
Select line 40 and invoke the **Source Pane** popup menu.	Triple-click with the first mouse button on line 40.
Set the breakpoint.	Hold down the third mouse button to invoke the popup menu, then choose **Toggle Line Break**.

Fx will mark the location of the breakpoint with the letter B in the **Source Pane**. When this line is encountered during program execution, the program will stop.

Setting a Breakpoint within a Procedure

The **Break In** menu item in the **Source Pane** popup menu sets a breakpoint within a procedure. The second breakpoint we will set is in the procedure `GetCounts` called from line 63. This breakpoint will stop the program when `GetCounts` is called.

To...	*Do...*
Select the procedure and invoke the **Source Pane** popup menu.	Select `GetCounts` and hold the third mouse button.
Set the breakpoint in `GetCounts.`	Choose **Break In**.

Be careful: you must make sure that you select only the name of the procedure. Selecting any other characters will result in an error.

Now that the two breakpoints are set, the program can be executed to explore other features of the debugger.

Executing Programs

The Fx Debugger can execute a program in different ways. The **Run…** menu item, located in the Execute menu, starts program execution. This menu item invokes a dialog which allows program arguments to be specified.

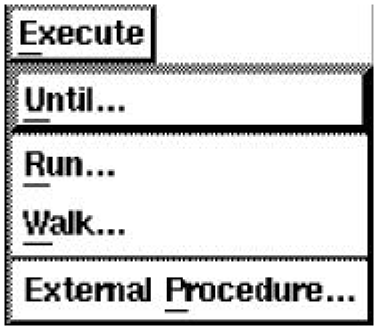


Figure 2-4
The Execute menu

Using the Run… Menu Item

To execute the `wrdcnt` program, a file needs to be specified so the program can count words, lines and characters. After choosing the **Run…** menu item, this file can be specified as an argument in the **Run Program** dialog. For the tutorial, the number of lines, words and characters in the `wrdcnt.c` source file will be counted.

To...	*Do...*
Execute the `wrdcnt` program with the desired argument.	Choose the **Run…** menu item from the Execute menu to invoke the **Run Program** dialog. Enter `wrdcnt.c` in the **Program Argument** text field then click on the **Run** button.

The `wrdcnt` program will execute to the breakpoint we set at line 40 in the source code.

Printing Values of Variables

The values of program variables can be tracked during program execution and used as an aid to debugging a program. The **Print** menu item in the **Source Pane** popup menu will display the values of variables in the **Output Pane**.

To...	*Do...*
Print the value the variable `argc`.	Select the variable `argc` in line 40. Press and hold the third mouse button to invoke the popup menu and choose **Print**.

Fx displays the number of arguments passed to the program in the **Output Pane**. Now, execute the program to the next breakpoint.

To...	***Do...***
Execute the program to the next breakpoint.	Click the **Continue** button.

After execution stops, note the **Status Field** under the Menu Bar. It will show the location where you are currently working, including the line number, unit and file name.

Executing Single Statements

To execute one statement at a time, use the **Step Into** button in the main window. This button will execute each statement, including any procedures that the statement contains. The **Step Over** button, will also execute one statement, but if the line contains a procedure, the debugger will not stop program execution until the procedure stops executing. At this point in the tutorial, you should be at line 134.

To...	***Do...***
Execute the program to line 139.	Click the **Step Into** button to reach line 139.
Move to line 140 without stepping to the procedure in line 139.	Click the **Step Over** button.

If you executed line 139 by mistake and want to return to line 140, press the **Return** button to complete execution of the procedure.

Dereferencing a Pointer Variable

The **Print*** menu item in the **Source Pane** popup menu dereferences C pointer variables.

To...	***Do...***
Dereference the pointer variable `f` referenced in line 140.	Select the variable `f`, then press and hold the third mouse button to invoke the popup menu. Choose **Print***.

Fx displays the number of words, lines and characters counted to this point in the **Output Pane**.

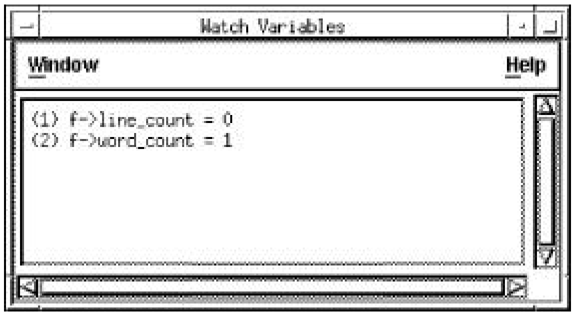
Watching Variables

Often, you will need to watch the value of a variable to determine if a procedure or function is overwriting it. The **Watch** menu item in the **Source Pane** popup menu allows you to watch the values of variables as the program executes.

To...	*Do...*
To watch the value of the expression `f->line_count`	Select `f->line_count` in line 135. Invoke the **Source Pane** popup menu and choose **Watch**.
Add `f->word_count` to the **Watch Variables** window.	Select `f->word_count` in line 136. Invoke the **Source Pane** popup menu and choose **Watch**.
Execute the program and watch the values of the variables as the statements are executed.	Click the **Step Over** button to execute the source lines. The values will be printed as the program executes.

Removing Watched Variables

To remove the variables that you are watching, use the Window menu within the **Watch Variable** window.



```
(1) f->line_count = 0
(2) f->word_count = 1
```

Figure 2-5
Watch Variables window

To...	*Do...*
Remove `f->word_count` from the **Watch Variables** window.	Select `f->word_count` in the **Watch Variables** window. Choose **Unwatch** from the Window menu.
Remove the other watched variable and close the **Watch Variables** window.	Choose **Close** from the Window menu.

Note: When all of the variables in the **Watch Variables** window are removed, the window will close automatically.

Changing Variables

Changing the value of a variable is done through the **Change** menu item in the **Source Pane** popup menu.

To...	*Do...*
Set the value of `f->word_count` to 100.	Select `f->word_count` and invoke the **Source Pane** popup menu. Choose **Change** to invoke the **Change Variable** dialog. Enter `100` in the **New Value** text field and the **Change** button

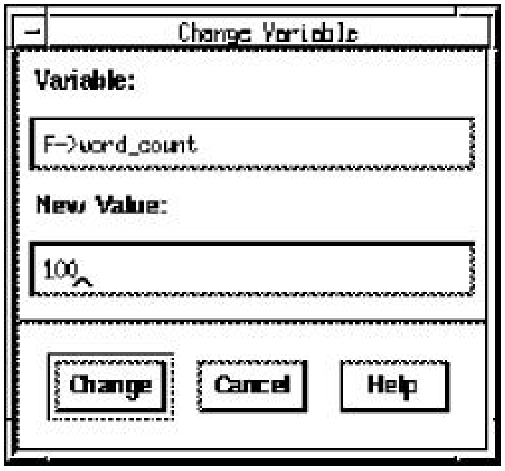


Figure 2-6
Change Variable dialog

If desired, you can display the new value of `f->word_count` by invoking the **Source Pane** popup menu and choosing **Print**.

Exiting the Debugger

You can end a debugging session at any time by choosing **Exit** from the File menu. Fx will terminate the program you are debugging and return you to the shell prompt.

To...	*Do...*

Exit from the debugger.	1. Choose **Exit** from the File menu.

Chapter Three

Fx Interface Reference

This chapter concentrates on the specific features of the Fx interface to enable you to get the most out of the debugger for your needs.

- **Using the Fx Interface**

  Describes using the Fx Debugger interface and its window: buttons, menus, and popup menus.

- **Reference Section**

  Provides a reference guide to the features of the debugger.

Using the Fx Interface

The Fx graphic interface provides a complete windowed environment for source level and assembly language debugging. The interface is based on the Open Software Foundation's OSF/Motif™ user environment, and its behavior is consistent with other compliant applications and with the OSF/Motif™ Window Manager.

If you have never used a graphical user interface before, you may wish to purchase additional documentation on the OSF/Motif™ user environment. One source of this information is the "OSF/Motif™ User's Guide" published by Prentice Hall, which is often included with UNIX documentation sets.

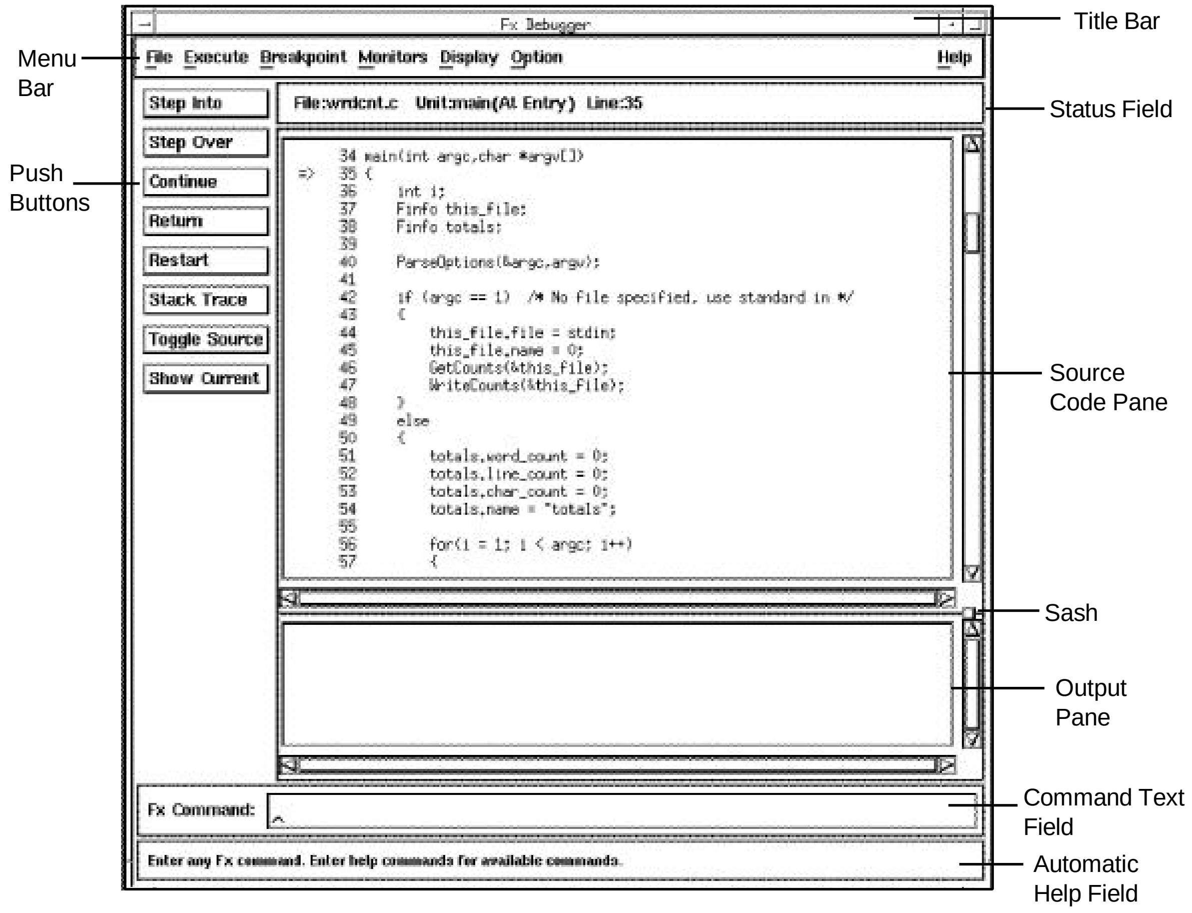
The Fx graphic interface permits all commands to be selected from within the debugger windows. Most commands are executed from the main window, the Fx Debugger window. Commands can be invoked by pressing buttons or using pull down menus and popup menus. The following are tips when working the interface:

- Any menu item that is followed by an ellipse (…) after the item name invokes a dialog.
- A menu or menu item may have a mnemonic, a short cut to accessing the menu from the keyboard. Using the underlined character in the menu name with the Alt key specifies the mnemonic. For example, to display the Option menu and its commands, press the Alt key and the letter O on the keyboard.
- A menu item can have a mnemonic or an accelerator. To select a menu item using its mnemonic, the menu must be open and visible. On the other hand, an accelerator is a key combination next to the menu selection that can be invoked

when the menu is not selected. For example, to set a breakpoint using its accelerator, press the Control key and the letter B.

Fx Main Window

The main window, the **Fx Debugger** window contains menus, buttons and text fields to view the source code and execute all debugging commands.



```
File:wrdcnt.c   Unit:main(At Entry)  Line:35
```

```
        34 main(int argc,char *argv[])
=>      35 {
        36     int i;
        37     Finfo this_file;
        38     Finfo totals;
        39
        40     ParseOptions(&argc,argv);
        41
        42     if (argc == 1)  /* No file specified, use standard in */
        43     {
        44         this_file.file = stdin;
        45         this_file.name = 0;
        46         GetCounts(&this_file);
        47         WriteCounts(&this_file);
        48     }
        49     else
        50     {
        51         totals.word_count = 0;
        52         totals.line_count = 0;
        53         totals.char_count = 0;
        54         totals.name = "totals";
        55
        56         for(i = 1; i < argc; i++)
        57         {
```

```
Enter any Fx command. Enter help commands for available commands.
```

Figure 3-1
Fx Debugger window

Menus

Fx has seven main menus that invoke debugger commands. These menus are located under the Title Bar and provide easy access to Fx commands.

Status Field

Under the Menu Bar is the **Status Field** that shows information about the program you are debugging: the line number, file name and program unit.

Push Buttons

Eight push buttons are located on the left side of the window which invoke commonly used commands. To activate a button, point to the button and press the first button on the mouse.

Navigating Through Text

The window also contains two scrollable panes. The first is the **Source Pane**, which allows you to view the program code by using the scroll bars. The second is the **Output Pane** that displays output from Fx commands, such as printing the value of variables. The text in this pane will consume memory as it builds up, so it is a good idea to delete the output when it is not needed. The square at the top of the scroll bar on the **Output Pane**, a **Sash**, allows you to resize the window as needed.

Popup Menus

The two popup menus, the **Source Pane** popup menu and the **Output Pane** popup menu, are similar to pull-down menus except they can be activated anywhere within the **Source Pane** or the **Output Pane**. Pressing and holding the third mouse button activates the popup menus.

Command Entry Text Field

The bottom of the window contains a text field labeled, Fx Commands. All commands in Fx have a command line equivalent that can be entered and executed from this text field. Also there are some commands in Fx (a very limited number) that do not have a menu or button equivalent, and can only be executed from this text field.

The next sections describe all the features of the Fx Debugger. A summary of the menus, buttons and fields is given, followed by a description of the menu items.

Menus

To open a menu, position the cursor on the menu name and press the first mouse button. To execute a menu item, click on the item. A menu may contain an item with an arrow to the right of the menu item: this arrow denotes a cascade menu, or submenu. The submenu contains a group of like menu items.

Some menu items may have diamonds or boxes to the left side. The diamond next to a menu item indicates the active setting of two or more choices in the menu or submenu. The shaded box acts as an on/off switch and notes that the menu item is active.

Also, some menu items may invoke dialogs and windows. Most of the dialogs have a **Cancel** button to dismiss the dialog without performing an action and a **Help** button to provide more information.

File Menu

The File menu contains general menu items that control the Fx graphical interface and debugging sessions. From this menu a debugging session can be logged, reinitialized or shut down, in addition to other session controls.

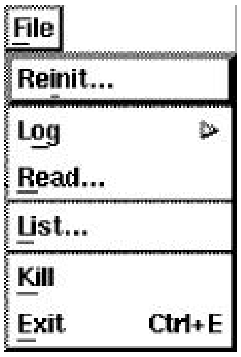


Figure 3-2
File menu

Reinit…

The **Reinit…** menu item restarts a debugging session without having to shut down. This is useful when specifying different source paths for a program already running, or when loading a new program. When the **Reinit…** menu item is invoked the following dialog appears:

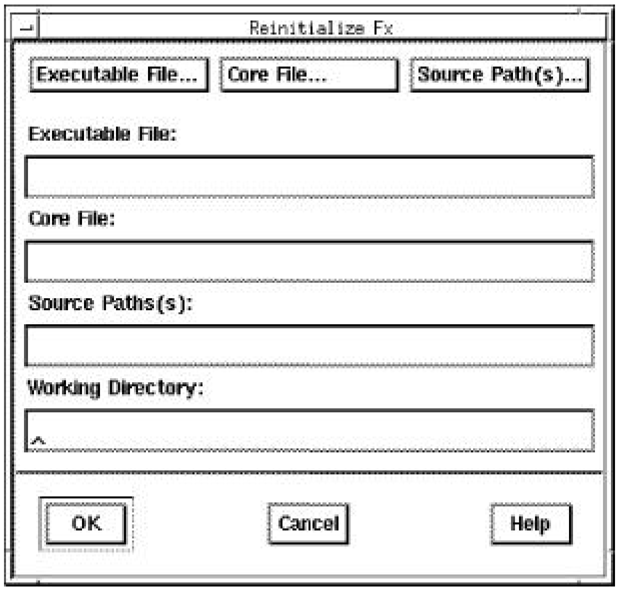


Figure 3-3
Reinit dialog

This dialog is similar to the **Load Program** dialog: use it to specify the executable file, core file, source path(s), and the working directory of the program you wish to debug. The four text fields will retain their original arguments if left unchanged. Press the **Cancel** button to dismiss the dialog without taking any action. If you need help or additional information about the dialog, press the **Help** button.

Log

The menu item contains a submenu with three possible selections: **File**, **On**, and **Off**. This submenu is used to create a separate file of debugger commands.

File…

Invokes a dialog to create and name a log file. The name of the file is entered into the **File Name** text field. When the **OK** button is pressed, the debugger will record all subsequent commands into the file.

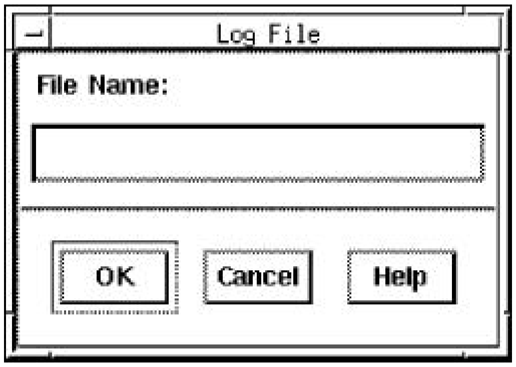


Figure 3-4
Log File dialog

On

Turns on the log function for a log file that has already been created.

Off

Shuts off the log function.

Read…

This menu item invokes a dialog allowing you to choose a file of debugging commands created using the **Log** menu item. When the **OK** button is pressed, the commands from the log file are executed.

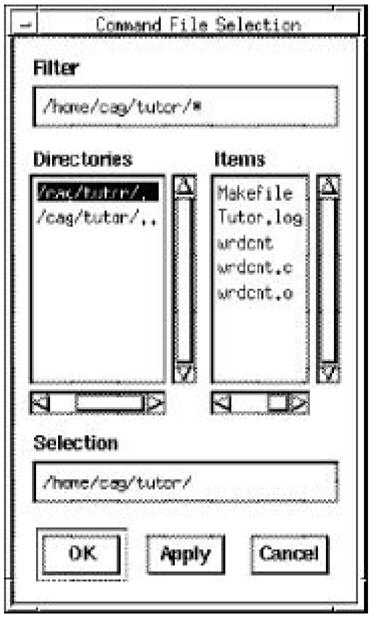


Figure 3-5
File Selection dialog

List…

When selected, the **List**… menu item invokes a file selection dialog to examine a file without leaving the debugger. To view a file, highlight the file and press the **OK** button.

Kill

The **Kill** menu item terminates the current process being debugged without exiting from the debugger. This menu item is useful, especially when you need to recompile a program without leaving the debugger. It is not necessary to use the **Kill** menu item when you exit from the debugger or restart program execution with the **Run…** menu item: both the **Exit** and the **Run…** menu items kill the current process before they execute.

Exit

This menu item closes the Fx Debugger interface and returns you to your shell.

Execute Menu

The Execute menu contains menu items that execute program code during a debugging session.

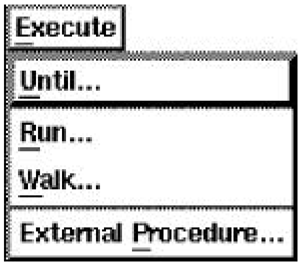


Figure 3-6
Execute menu

Until…

Executes the program to a certain location. This menu item will show the following dialog when selected.

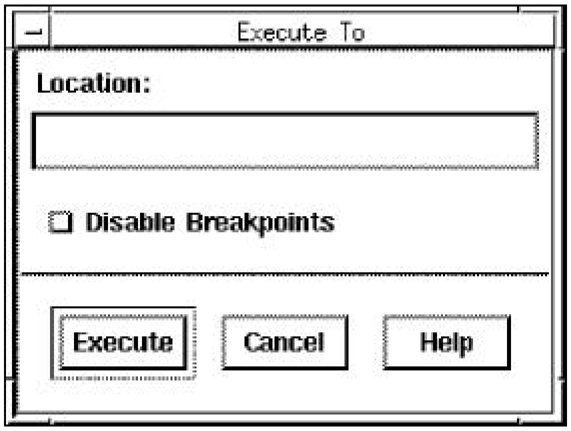


Figure 3-7
Execute To dialog

The **Execute To** dialog can be used to run a program to a certain location in the code. When a location is specified in the **Location** text field, the debugger sets a temporary breakpoint. The **Disable Breakpoints** check box tells the debugger to ignore all other breakpoints set with the Breakpoint menu.

Run…

The **Run…** menu item is used to start or restart execution of a program.

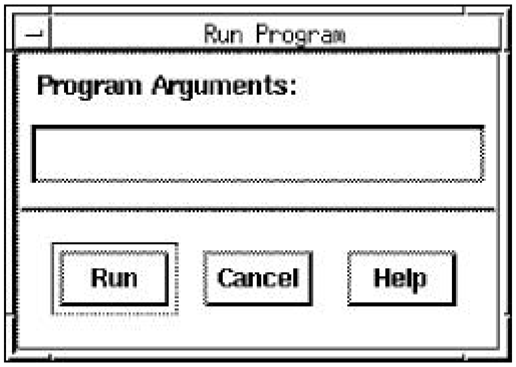


Figure 3-8
Run Program dialog

It can be used to pass arguments to the program, such as redirecting the standard input and output while the program executes.

Walk…

The **Walk…** menu item allows other debugger commands to be repeatedly issued by the debugger itself. The command, or a series of commands, to be invoked is specified in the **Walk Command(s)** text field.

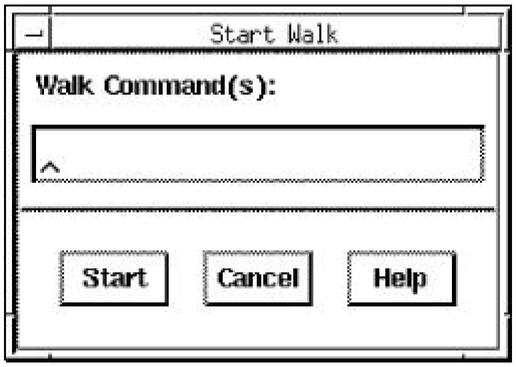


Figure 3-9
Start Walk dialog

After the **Walk** command is issued with the **Start** button, the debugger will continue to execute the specified commands. The **Start Walk** dialog is replaced by the **Stop Walk** dialog. To stop the command at any time, press the **Stop** button.

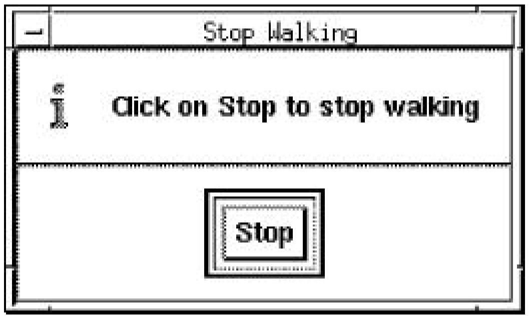


Figure 3-10
Stop Walk dialog

External Procedure…

The **External Procedure…** menu item allows procedures and functions within the program to be specified and executed out of sequence.

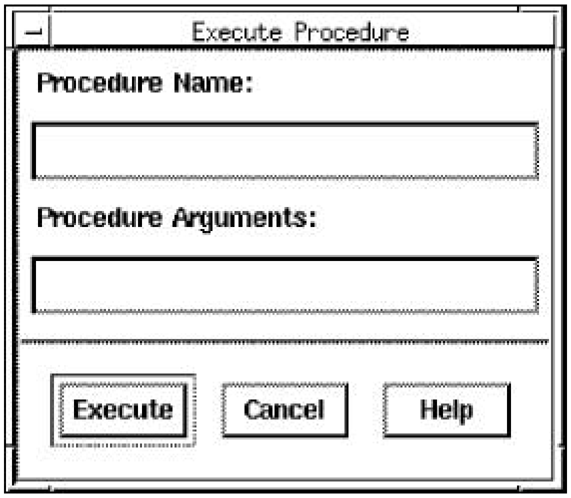


Figure 3-11
External Procedure dialog

Press the **Execute** button to execute the procedure or function. Press **Cancel** to dismiss the dialog without executing a procedure. To get additional information, press the **Help** button.

Breakpoint Menu

The Breakpoint menu contains menu items for setting, listing and deleting breakpoints. Breakpoints are analogous to stop signs: when a breakpoint is encountered during program execution, the program suspends execution and returns control to the debugger. They are useful for analyzing sections of code and can be set at a particular line, subroutine or function. Breakpoints are covered in detail in Chapter 5.

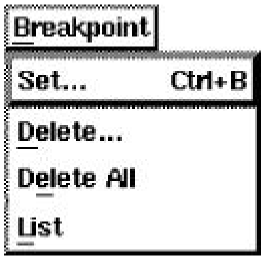


Figure 3-12
Breakpoint menu

Set…

The **Set…** menu item invokes the **Set Breakpoint** dialog for placing a breakpoint in a given location. To dismiss the dialog without setting a breakpoint, press the **Close** button.

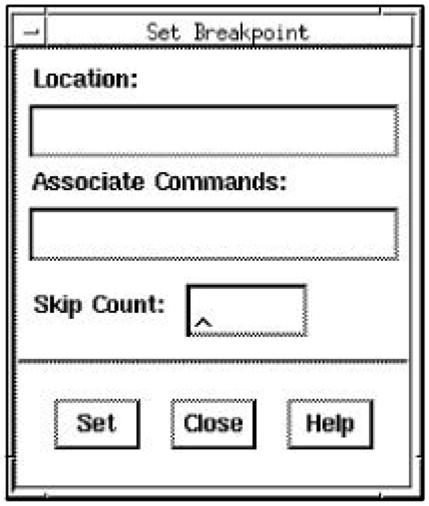


Figure 3-13
Set Breakpoint dialog

Location Text Field

The location of the breakpoint is entered into this text field. The location of a breakpoint can be at any valid address expression, a line number, or the first executable source line in a procedure. For more information on how to specify a breakpoint location, see Chapter 5.

Associate Commands Text Field

Specifies the debugger commands to be executed when the breakpoint stops program execution. When execution stops at the specified breakpoint, the commands will be executed.

Skip Count Text Field

The skip count refers to the number of times the breakpoint is to be ignored before program execution is stopped. If a skip count is not specified, a value of zero will be used–program execution will stop the first time the breakpoint is encountered.

Delete…

The **Delete…** menu item is used to remove one or more breakpoints from a program. When selected, the command will display the **Delete Breakpoints** dialog, shown below:

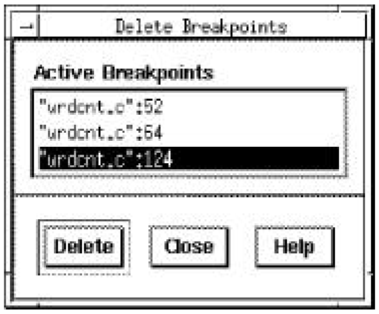


Figure 3-14
Delete Breakpoints dialog

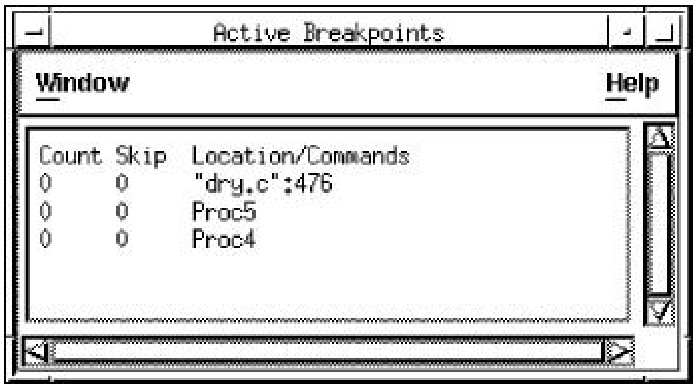
To delete a breakpoint from the list, highlight the breakpoint and press the **Delete** button.

Delete All

The **Delete All** menu item removes all breakpoints from a program.

List

The **List** menu item invokes a dialog that shows all active breakpoints, any associated commands, skip counts, and a count of the number of times each breakpoint has been encountered without halting program execution.



```
Count Skip  Location/Commands
0     0     "dry.c":476
0     0     Proc5
0     0     Proc4
```

Figure 3-15
List Breakpoint window

Monitors Menu

The Monitors menu contains menu items for setting, removing, and listing variable monitors. Invoking a monitor is similar to setting a breakpoint, with a notable exception. When a monitor is set, it will not halt execution at a specific location. Instead execution halts when a relational expression specified by the programmer evaluates true.

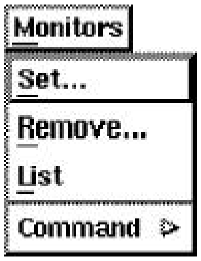


Figure 3-16
Monitors menu

Set…

The menu item invokes the **Set Monitor** dialog. The text fields create a relational expression that prompts the debugger to stop execution when a variable changes value.

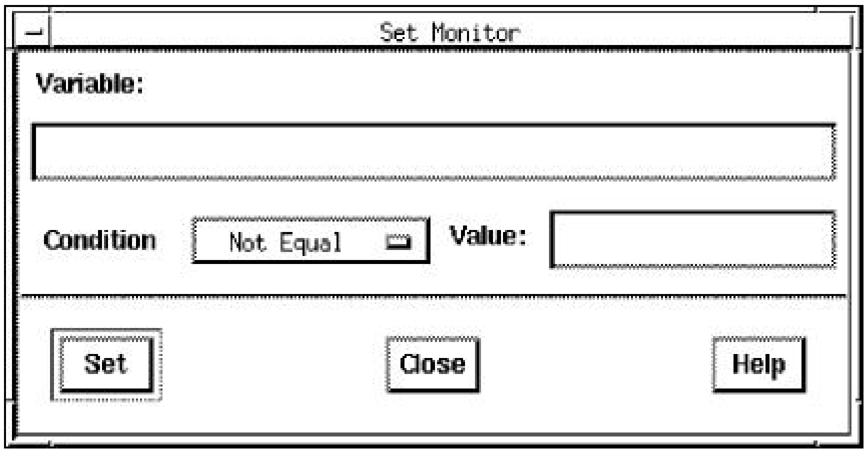


Figure 3-17
Set Monitors dialog

Variable Text Field

Specifies the name of the variable where the monitor will be set.

Condition Option Menu

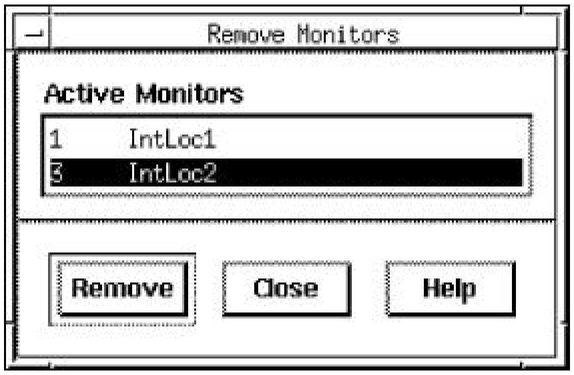
Determines the operator used to compare the variable with the value specified in the **Value** text field.

Value Text Field

Contains the value of the variable at which execution will stop.

Remove…

The **Remove…** menu item removes a monitor previously installed.

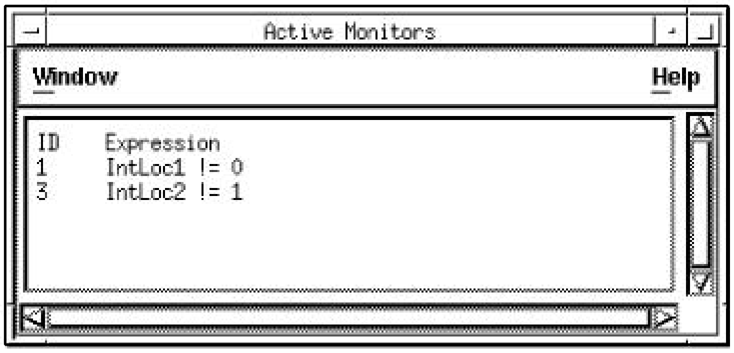


```
1      IntLoc1
3      IntLoc2
```

Figure 3-18
Remove Monitors dialog

List

The **List** menu item displays all active monitors with ids and expressions.



```
ID    Expression
1     IntLoc1 != 0
3     IntLoc2 != 1
```

Figure 3-19
List Monitors window

Command

The **Command** submenu allows you to select how Fx will execute the **Continue** command when working with monitors. A diamond will mark the current selection.

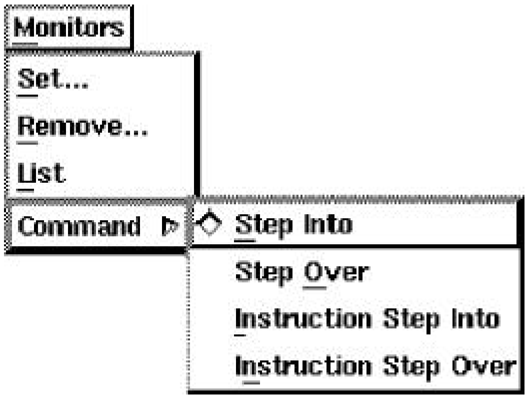


Figure 3-20
Monitors menu with command submenu

Step Into

The **Step Into** item specifies that when the **Continue** command is invoked, the next statement is executed and the value of the monitor will be checked.

Step Over

The **Step Over** item specifies that when the **Continue** command is invoked, the next statement is executed and the value of the monitor will be checked. If the statement that is executed contains any procedures, they are treated as part of the statement line

Instruction Step Into

The **Instruction Step Into** item specifies that when the **Continue** command is invoked while working in assembly level, the next instruction is executed and the value of the monitor will be checked.

Instruction Step Over

The **Instruction Step Over** item specifies when the **Continue** command is invoked while working in assembly level, the next instruction executes and the value of the monitor will be checked. If the instruction is a call to a procedure, it will be treated as a single statement.

Display Menu

The Display menu contains menu selections for controlling and printing information about the program.

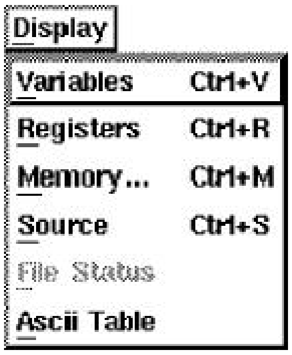
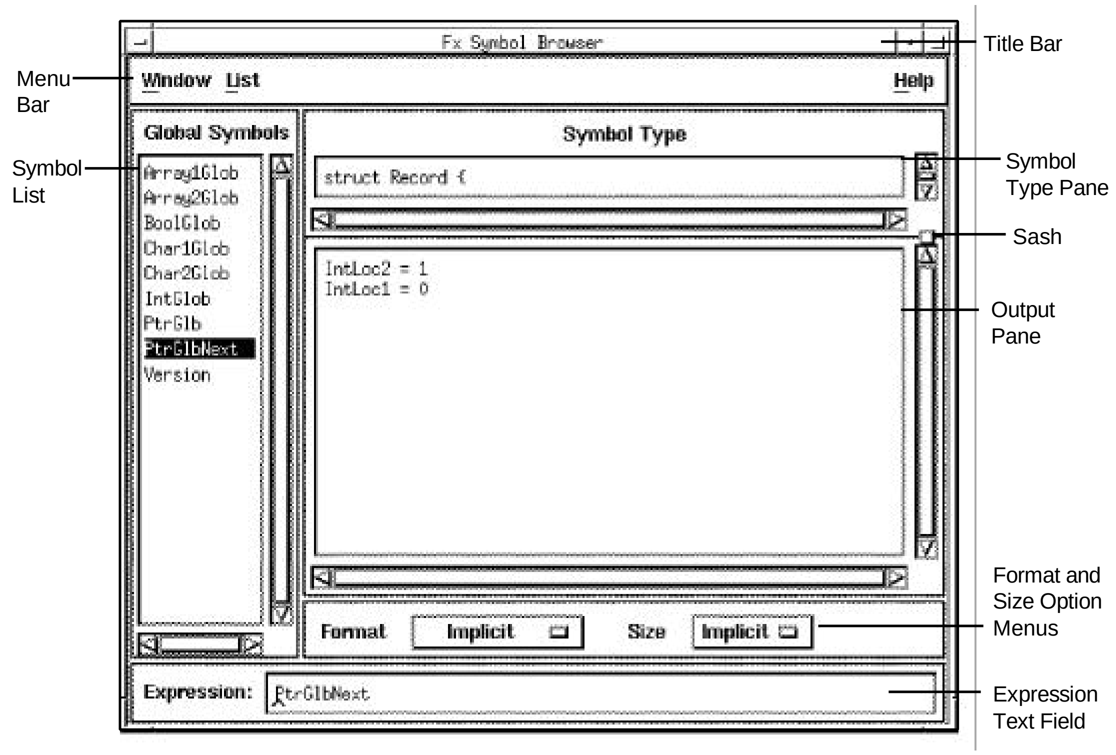


Figure 3-20
Display menu

Variables

The **Variables** menu item allows the programmer to display global, static or local variables. When selected, the **Variables** menu item invokes the **Symbol Browser**, shown below. For more information on using the **Symbol Browser**, see Chapter 6, “Working with Variables.”



```
Array1Glob
Array2Glob
BoolGlob
Char1Glob
Char2Glob
IntGlob
PtrGlb
PtrGlbNext
Version
```

```
struct Record {
```

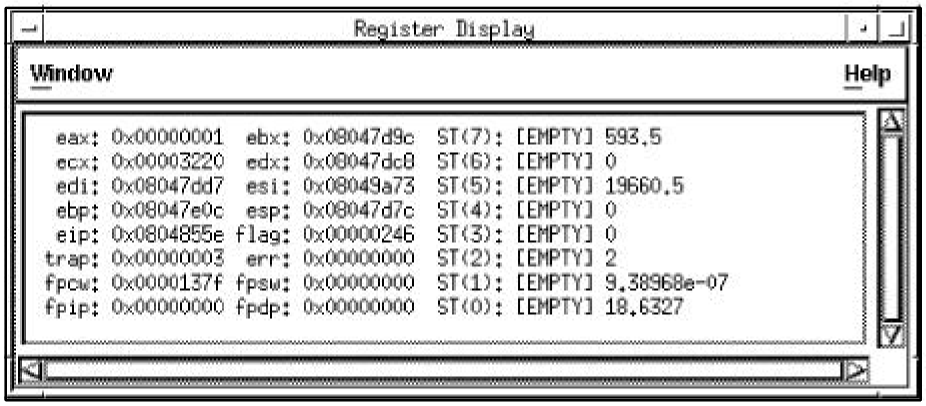
```
IntLoc2 = 1
IntLoc1 = 0
```

```
PtrGlbNext
```

Figure 3-21
Symbol Browser Window

Registers

The **Registers** selection is used to display the contents of the machine registers.

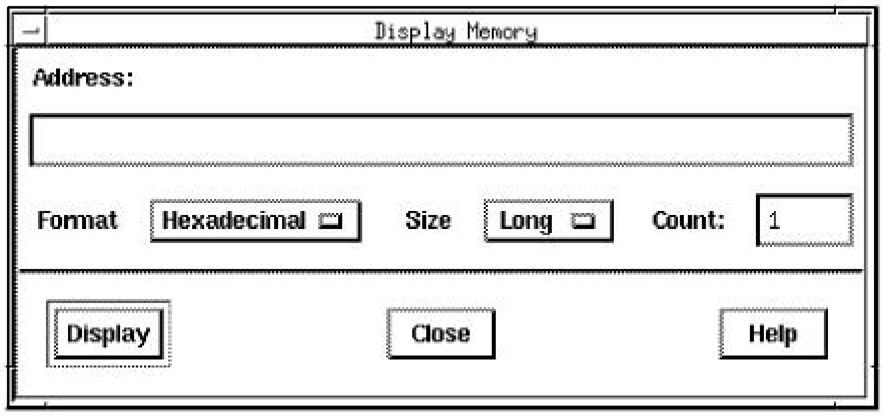


```
  eax: 0x00000001  ebx: 0x08047d9c  ST(7): [EMPTY] 593.5
  ecx: 0x00003220  edx: 0x08047dc8  ST(6): [EMPTY] 0
  edi: 0x08047dd7  esi: 0x08049a73  ST(5): [EMPTY] 19660.5
  ebp: 0x08047e0c  esp: 0x08047d7c  ST(4): [EMPTY] 0
  eip: 0x0804855e flag: 0x00000246  ST(3): [EMPTY] 0
 trap: 0x00000003  err: 0x00000000  ST(2): [EMPTY] 2
 fpcw: 0x0000137f fpsw: 0x00000000  ST(1): [EMPTY] 9.38968e-07
 fpip: 0x00000000 fpdp: 0x00000000  ST(0): [EMPTY] 18.6327
```

Figure 3-22
Register Display window

Memory…

This menu item is used to select memory addresses beginning at a specific location. The address is specified in the **Display Memory** dialog, and listed in the **Dump Memory** window that appears when the **Display** button is pressed.



Address:

Format Hexadecimal Size Long Count: 1

Figure 3-23
Display Memory dialog

Address Text Field

The address should be entered into this text field. For more information on proper syntax for this text field, see Chapter 6.

Format Option Menu

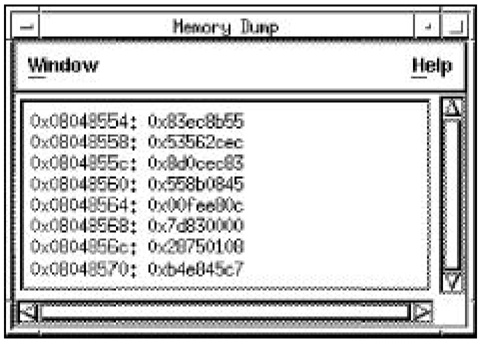
This menu denotes how the memory will be displayed.

Size Option Menu

This menu denotes memory size.

Count Text Field

This text field specifies the number of memory locations displayed in the **Memory Dump** window.

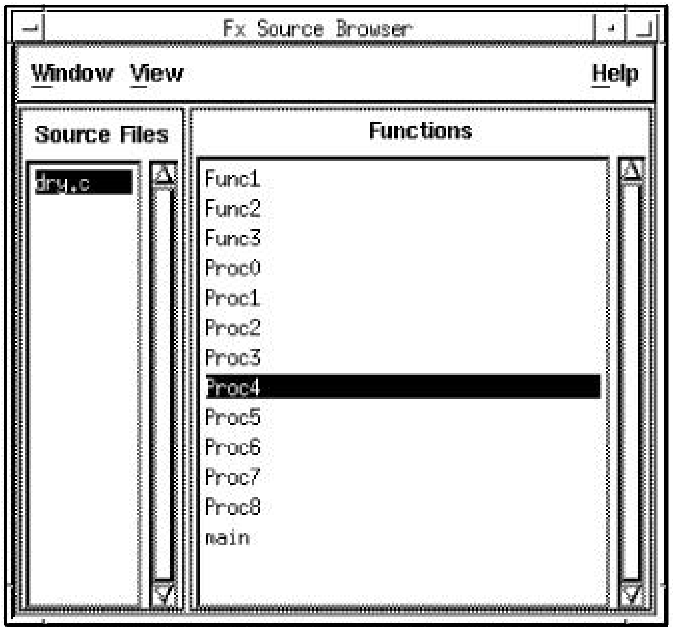


```
0x08048554: 0x83ec8b55
0x08048558: 0x53562cec
0x0804855c: 0x8d0cec83
0x08048560: 0x558b0845
0x08048564: 0x00fee00c
0x08048568: 0x7d830000
0x0804856c: 0x20750100
0x08048570: 0xb4e045c7
```

Figure 3-24
Memory dump window

Source

The **Source** menu item invokes the **Source Browser** window.



dry.c

Func1
Func2
Func3
Proc0
Proc1
Proc2
Proc3
Proc4
Proc5
Proc6
Proc7
Proc8
main

Figure 3-25
Source Browser window

The **Source Browser** lists all the source files for the program that have been compiled with the **-g** option. When a source file is clicked, the functions within the file are listed on the right side of the window. Within the View menu, the **View Code** menu item will display the code in the **Fx Debugger** window.

File Status

Displays information about all connected and preconnected FORTRAN units. This menu item is active only when debugging programs compiled with Absoft compilers.

ASCII Table

This menu item invokes a table of ASCII characters and their corresponding hexadecimal numbers.

Option Menu

The Option menu contains two submenus with selections for customizing the way you view and work with source code.

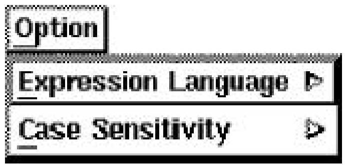


Figure 3-26
Option menu

Expression Language

This submenu has three selections, **Automatic**, **C**, or **FORTRAN**, to choose the language in which expressions are evaluated. The default is set to **Automatic**: the expression language will automatically correspond to the current source code file.

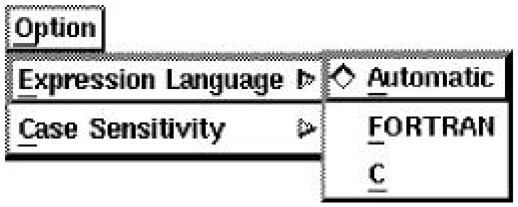


Figure 3-27
Expression Language submenu

Case Sensitivity

This submenu allows you to set the case sensitivity for variable and procedure names, since some compilers may fold all upper case letters into lower case when compiling code.

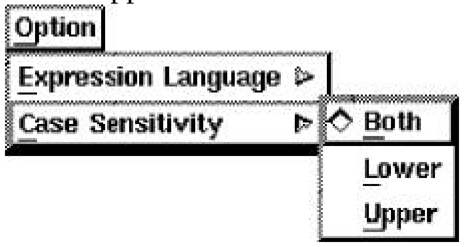


Figure 3-28
Case Sensitivity submenu

Help Menu

The Help menu allows you to access additional information on the features of the Fx Debugger. The on-line help system displays information on commands and windows. The help system is composed of two parts: the **Automatic Help Field** and the **Help Window**.

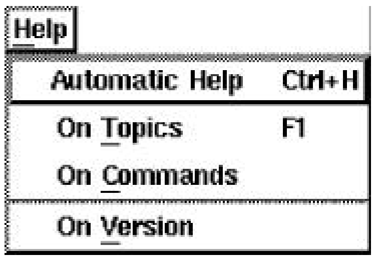


Figure 3-29
Help menu

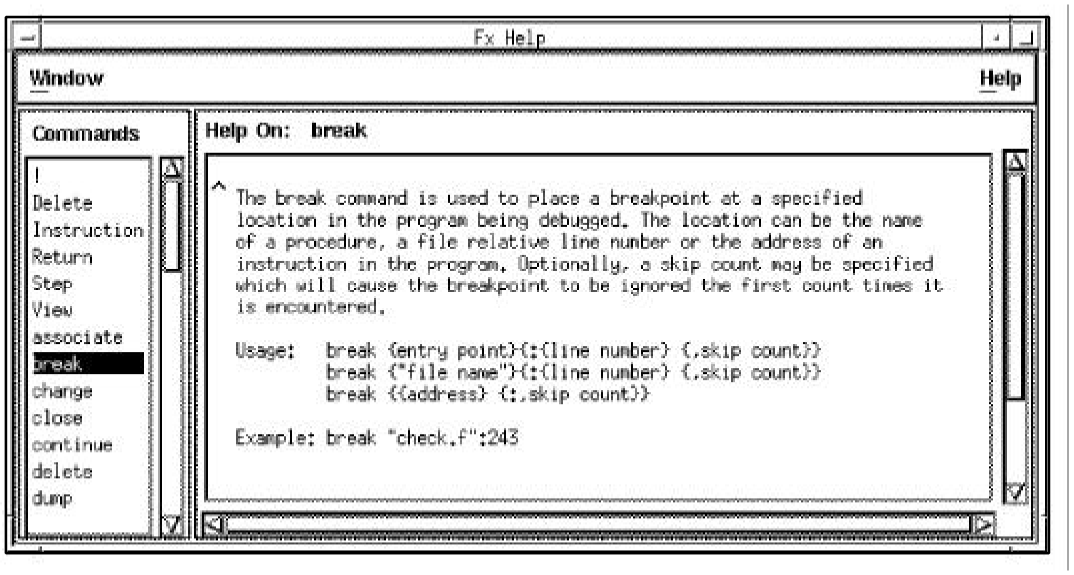
Automatic Help Field

The **Automatic Help Field** is located at the bottom of the main window. When you begin a debugging session, this window will display information about the version of Fx that you are using. You can toggle off the **Automatic Help Field** by choosing the **Automatic Help** item in the Help menu.

You can display additional information about the buttons, text boxes and menus in the text field by moving the mouse pointer over an item and pressing the left mouse button. The action performed by a button or menu will not be activated until you release the mouse button. If you do not wish to perform the specified action, simply move the mouse pointer off the item before releasing the button.

Using the Help Window

The help window is activated by selecting the **On Topics** item or the **On Commands** item from Help menu in the Fx Debugger window.



```
The break command is used to place a breakpoint at a specified
location in the program being debugged. The location can be the name
of a procedure, a file relative line number or the address of an
instruction in the program. Optionally, a skip count may be specified
which will cause the breakpoint to be ignored the first count times it
is encountered.

Usage:   break {entry point}{:{line number} {,skip count}}
         break {"file name"}{:{line number} {,skip count}}
         break {{address} {:,skip count}}

Example: break "check.f":243
```

Figure 3-30
Help window

After the window is visible, you can display additional information by choosing an item from the list of topics or commands.

On Topics

Activates the help window with information on topics of interest.

On Commands

Activates the help window with information on commands available in Fx.

On Version

Display a dialog showing the version of the Fx Debugger installed.

Popup Menus

The **Source Pane** popup menu and the **Output Pane** popup menu contain menu items for working with selections in the **Source Pane** and the **Output Pane.**

Source Pane Popup Menu

The **Source Pane** popup menu contains menu items for setting breakpoints, viewing code for procedures and printing variables in the **Source Pane**. The menu is invoked by selecting source code text and pressing the third mouse button.

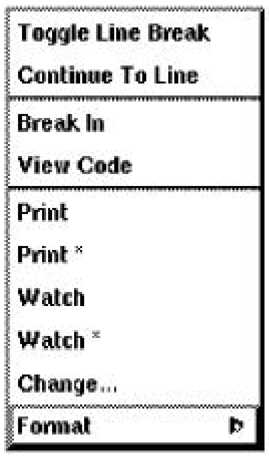


Figure 3-31
Source Pane popup menu

Toggle Line Break

This menu item sets or deletes breakpoints in the program.

Continue To Line

Allows the program to execute to a certain line. This is equivalent to setting a temporary breakpoint that will disappear when execution halts.

Break In

This menu item sets a breakpoint within a function or procedure. When set the breakpoint will halt program execution at the first executable statement in the procedure.

View Code

Allows you to view the code for a procedure selected in the **Source Pane**.

Print

Prints the value of a selected variable to the **Output Pane**.

Print*

This menu item dereferences a selected pointer variable and prints the result to the Output Pane.

Watch

The **Watch** menu item invokes the **Watch Variables** window, from which you can watch the value of variables change during program execution.

Watch*

The **Watch*** menu item invokes the **Watch Variables** window, from which you can watch dereferenced pointers during program execution.

Change…

The **Change…** menu item invokes the **Change Variable** dialog. Using this dialog, the value of a selected variable can be changed by entering the new value into the **New Value** text field and pressing the **Change** button.

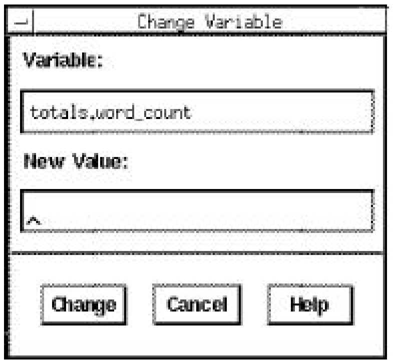


Figure 3-32
Change Variable dialog

Format

This selection contains a submenu that allows you to choose how output will be formatted in the **Output Pane**.

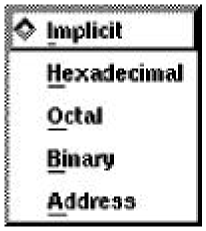


Figure 3-33
Format submenu

Output Pane Popup Menu

The six menu items in the **Output Pane** popup menu are similar to the **Source Pane** popup menu, except they apply to the text selected in the **Output Pane**.

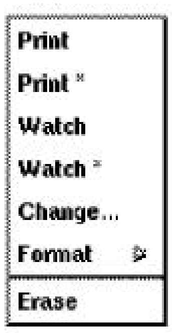


Figure 3-34
Output Pane popup menu

Print

This selection prints the value of the variable selected at the bottom of the **Output Pane**.

Print*

This menu item dereferences a pointer and prints the output to the **Output Pane**.

Watch

The **Watch** menu item invokes the **Watch Variables** window to watch the value of variables as they change during program execution.

Watch*

This menu item invokes the **Watch Variables** window, from which you can watch dereferenced pointers during program execution.

Change…

The **Change…** menu item invokes the **Change Variable** dialog. In this dialog, the desired value can be entered into the **New Value** text field.

Format

The **Format** menu item contains a submenu that designates the format of output in the **Output Pane**.

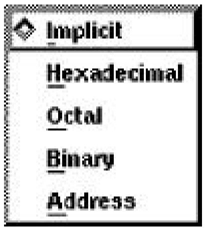


Figure 3-35
Format submenu

Erase

The **Erase** menu item clears all text from the **Output Pane.**

Command Push Buttons

The command push buttons, located on the left side of the **Fx Debugger** window, access frequently used commands.

Step Into

The **Step Into** button executes single statements. When pressed, the next line of code will be executed. If the source line contains a function or subroutine call, execution will stop at the first executable line of the function or subroutine.

Step Over

This button also executes single statements. However, it does not follow subroutine or function references, like the **Step Into** button, but rather stops on the next source line of the current procedure.

Continue

The **Continue** button resumes program execution until a breakpoint is encountered, an error occurs, or the program runs to completion.

Return

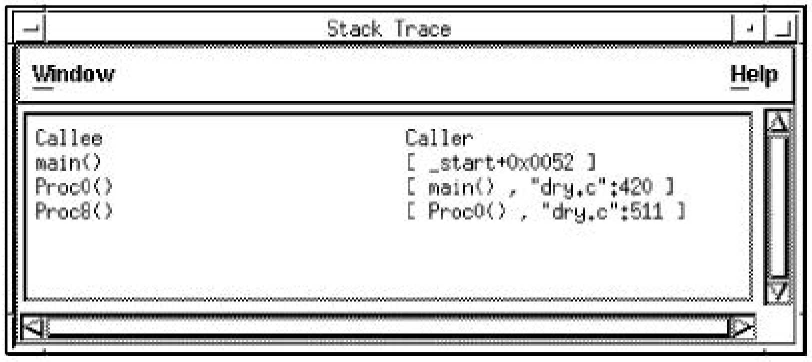
The **Return** button resumes program execution until the current procedure returns to its calling procedure.

Restart

The **Restart** button restarts program execution from the beginning of the program.

Stack Trace

This button invokes the **Stack Trace** window to allow you to trace all procedures previously executed. Once opened, this window will display the procedures as the program executes.



```
Callee                          Caller
main()                          [ _start+0x0052 ]
Proc0()                         [ main() , "dry.c":420 ]
Proc8()                         [ Proc0() , "dry.c":511 ]
```

Figure 3-36
Stack Trace window

Toggle Source

This button toggles the source code view between high-level language and assembly language.

Show Current

When pressed, this button displays the next line to be executed in the **Source Pane**.

Command Entry Text Field

Although Fx provides an excellent windowed interface to execute debugging commands from buttons and menus, commands can also be executed by using the **Command Entry** text field, located at the bottom of the **Fx Debugger** window. Although most of the commands are available to the user in the menus or with push buttons, this text field can be used as a quick alternative to issue a command to the debugger.

Using the Command Entry Text Field

To use the **Command Entry** text field, enter the name of the command and press the Return key. A command can be executed with arguments, and must follow the syntax format in the Command Reference in Chapter 9.

Previous commands executed from this text field can be reviewed by pressing the Up and the Down arrow keys. The **Command Entry** text field will recall up to 25 commands. If you press the Return key, the command will execute.

Chapter Four

Running Fx

This chapter introduces the basics of running the Fx debugger to debug programs. The following topics are covered in this chapter:

- **Preparing Your Program for Debugging**

  Discusses compile time options to take full advantage of the Fx Debugger.

- **Starting an Fx Debugging Session**

  Provides details on loading a program and changing the initialization file.

- **Restarting a Debugging Session**

  Introduces the Fx command for restarting a debugging session without leaving the debugger.

- **Debugging with Core Files**

  Describes core files and introduces ways to use core files to pinpoint and correct errors in source code.

- **Examining Source Code**

  Presents methods to examine multiple source files within a program, view procedures, and list procedure calls.

- **Recreating a Debugging Session**

  Shows how to use the log feature of Fx to record and playback commands initiated during a debugging session.

Preparing Your Program For Debugging

In order to debug at the source level, the **-g** option must be specified when compiling source code (for gcc compiled code on Linux, use **–gdwarf**). Specifying the **-g** option instructs compilers to include additional information in the executable program that describes the program's source files, procedures, and variables.

The -g Option

Use the **-g** option of the standard UNIX compilers to take full advantage of Fx features. Be aware that the **-g** option requires compilers, assemblers, and linkers to perform additional work, thus the process of compiling your code will be slower. In addition, the information that allows you to debug at the source level increases the amount of disk space required for your program.

For trivial programs this is not a consideration, but if the program consists of sixty or seventy source files you may want to note that Fx does not require that all of your source files be compiled with the **-g** option. When debugging large programs, you may wish to select a subset of your source files to compile with the **-g** option. For example, if you are attempting to add a new feature to a program, you might re-compile only the files that implement the new feature.

When preparing your program for debugging, you should keep in mind that many compilers will have options which adversely affect source level debugging. Compiler options that cause optimization, such as the **-O** option, should not be used while you are still debugging a program. Although it is not impossible to debug optimized code at the source level, optimizations can create additional problems. For more information, see Appendix C for information on debugging with optimized code.

Starting an Fx Debugging Session

This section gives information on loading programs for debugging with Fx.

Loading a Program for Debugging

Start Fx by entering `xfx` at the command shell prompt. Fx will display the **Load Program** dialog used to specify the executable file, core file, source path(s), and the working directory.

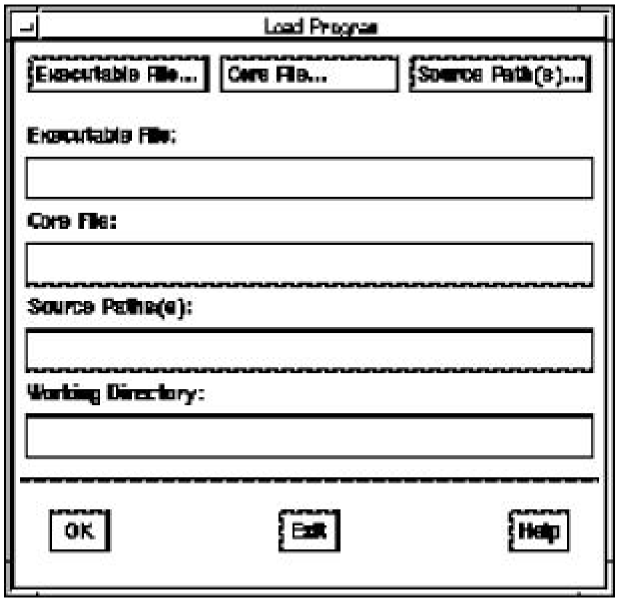


Figure 4-1
Load Program dialog

Executable File Text Field

The executable file is the program to be debugged. Since the standard compilers usually create an executable program named `a.out`, Fx will supply this default name. If you need to specify the file to be used, enter the name of the file in the box, or use the **Executable File** button to select the file from the selection box.

Core File Text Field

Fx allows you to use the core file of a program to determine the location of errors in the code. Fx will automatically use the core file if it exists in the same directory as the program being debugged. If the core file is in another location, you must specify it by typing its directory and name into the **Core File** text field. Core files are also discussed in the section entitled, "Debugging with Core Files" later in this chapter.

Source Path(s) Text Field

This text field allows you to specify the directory or directories that contain the source files for the program being debugged.

Working Directory Text Field

This text field specifies the directory that will become the program's current directory when running under Fx.

Using the Selection Dialogs

Along the top of the **Load Program** dialog there are three buttons that will invoke a file selection dialog which allows you to choose the executable file, the core file, and source paths. The dialog allows you to choose the files: select the file and click the **OK** button.

After specifying the necessary files, click the **OK** button in the **Load Program** dialog to start the debugger.

Executing Fx Commands During Initialization

You can also have Fx automatically execute commands after it has loaded your program. This section discusses the Fx initialization file that makes this possible.

About .fxinit

Each time you start a debugging session, Fx looks for a file named `.fxinit` in the current working directory and in the directory specified by the environment variable `HOME`. If this file is found, Fx will execute any commands it contains. If `.fxinit` exists in both directories, the file in the `HOME` directory will be read first.

The primary purpose of this file is to automatically redefine the internal variables within Fx, allowing you to customize Fx’s defaults for command behavior. These internal variables, referred to as control variables, are listed in Appendix B. An initialization file can also be used to execute any Fx command. For example, if you are repeatedly debugging the same procedures in a program, you might create an initialization file that will automatically set breakpoints on these procedures.

A sample .fxinit file

```
# Lines which start with the character '#' and blank lines
# are ignored, allowing comments to be added if desired.
# Set the format used for displaying single precision values
change $ffmt = "(E12.4E3)"
# Set the maximum number of array elements displayed when
# unsubscripted arrays are used with the print command
change $acount = 10
# Set the default expression language to FORTRAN
change $deflang = "FORTRAN"
# Set the default tab spacing
change $tabsize = 5
```

Restarting A Debugging Session

When you start a debugging session, you may realize that you failed to specify a directory containing files needed to debug your program. The **Reinit…** menu item is used to restart Fx and specify the new directories without terminating the debugging session.

The Reinit… Menu item

The **Reinit…** menu item allows you to change the command line arguments entered when Fx was first launched. The menu item invokes the Reinitialize dialog that allows you to change the name of the program being debugged, the name of the core file associated with the program, and the list of source directories.

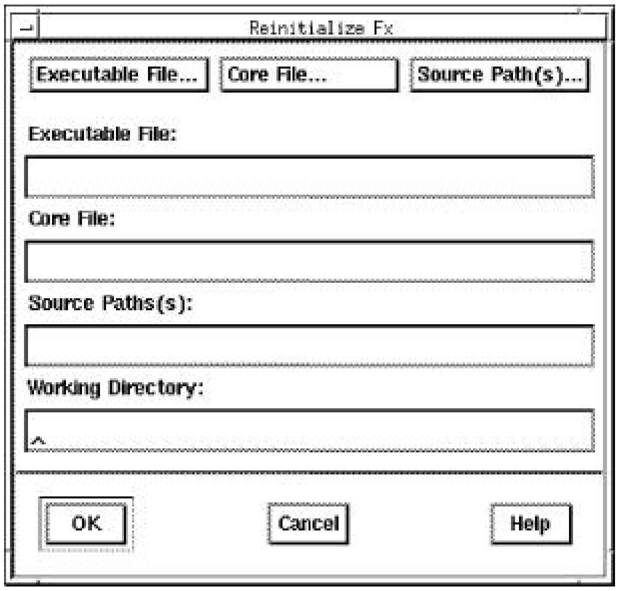


Figure 4-3
Reinitialize dialog

For example, if you started Fx and found that you forgot to specify the source path `/home/sysdep/`, you can restart the debugging session by doing the following:

Example:

- Choose **Reinit** from the File menu.
- Enter `/home/sysdep/` in the source path text field.
- Press the **OK** button.

Since the program name and the core file were not changed, Fx will continue to use the original arguments specified: `a.out` and `none`.

You can also use the **Reinit…** menu item to begin a debugging session on a completely different program. If you have finished debugging `a.out`, you can begin debugging another program using completely different parameters.

Debugging with Core Files

When running a program under development, a message similar to

```
Segmentation violation - core dumped
```

may appear. When this happens the operating system creates a file containing an image of your program's memory at the time the error occurred. This file, a core file, can be used in Fx to track the errors in the program.

Using Core Files

At the beginning of a debugging session, Fx looks in the current directory for the core file. If this file exists, Fx uses it to determine the location of errors and display the offending source line. Since your program is not executing when a core file is used, the menu items for executing programs will be grayed.

Often, a core dump will be produced by an error in one of the libraries linked with your program. This can occur because of incorrect arguments passed to a library routine or because there is a bug in the library itself. In either case, you are unlikely to have source code for the library and Fx will display a screen full of assembly language instructions. What you need to know is the point in your program where the library routine was called.

Using the **Stack Trace** button you can display a list of all the procedures in your program, starting with the main entry point, and ending with the procedure that caused the core dump. By examining this list, you can determine where the error occurred in your program.

When you have determined the location, one course of action is to set a breakpoint and start program execution with the **Run…** menu item. When the breakpoint is encountered, you can use the **Print** menu item to verify that you are passing the correct arguments to the routine that caused the error.

When a debugging session is initiated without a core file, Fx will load the program and execute it until its main procedure is entered. Further execution of the program under the debugger interface is controlled using the buttons and menus in the main window.

Examining Source Code

Source code can be viewed within the current source code file or within other source files.

Viewing Program Source Files

The Source Browser simplifies examining source code. With the **Source Browser** you can find the name of a source file, function or procedure. This is particularly useful when debugging code written by another programmer when you know the name of a procedure, but do not know its source file. The **Source Browser** is accessed through the **Source** menu item in the Display menu.

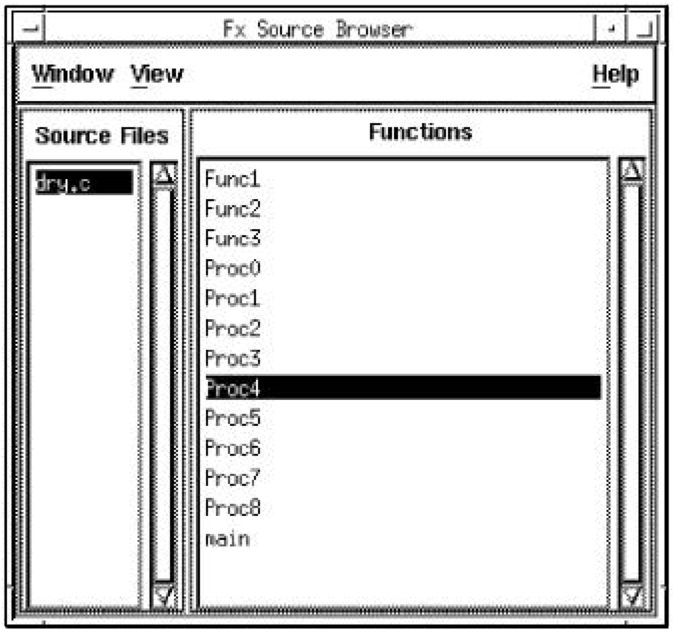


Figure 4-4
Source Browser window

The left side of the window will display the source files for a program when you choose the **Source** menu item from the **View** menu. Click on the name of a source file to view all of its functions. To display the source code for a function, click on the function in the Functions list box, and choose **View Function** from the **View** menu. You can also double-click on the function to see the source code in the main window.

When you want to return to the current source line, use the **Show Current** button in the main window.

Viewing Other Files

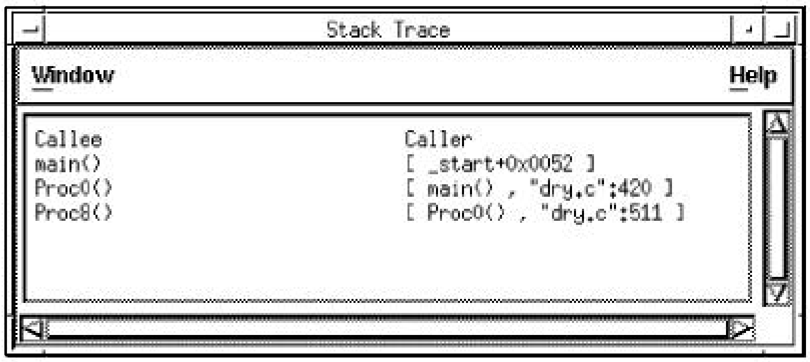
The **Source Browser** can only display files that comprise the source code for a program. To examine an include file to determine the value of a predefined constant or FORTRAN parameter, you can use the **List…** menu item under the File menu. This menu item allows you to examine the contents of any text file without leaving the debugger. When the **List…** item is invoked, the **List File Selection** dialog will appear. This dialog is like any other selection dialog; use it to select and open the include file or other text file.

Viewing Procedures

A procedure in the **Source Pane** can be viewed using the **Source Pane** popup menu. Select the procedure name within the statement and choose the **View Code** item. The window will show the code for the procedure.

Viewing Execution Status

The **Stack Trace** button in the **Fx Debugger** window is used to display the chain of procedure calls which produced the current program state. This button will invoke the **Stack Trace** window.



```
Callee                          Caller
main()                          [ _start+0x0052 ]
Proc0()                         [ main() , "dry.c":420 ]
Proc8()                         [ Proc0() , "dry.c":511 ]
```

Figure 4-5
Stack Trace window

In the window, each procedure in the current chain is listed along with the calling procedure that called it and the calling procedure's file and line number if available. Using the **View Code** menu item in the Window menu, a procedure may be selected and viewed in the **Source Pane.**

Recreating a Debugging Session

When debugging large programs, it often takes a considerable amount of work to get the program to the point where it fails. This process may involve entering many different Fx commands, or changing the values of program variables at different points in the program. Sometimes you will get the program to the point of failure and then accidentally execute the wrong command.

Fx provides the **Log** menu item to record all of the commands you have entered in a separate file and then play them back. This allows you to recreate the debugging session without having to reenter the commands.

Using the Log Selection

The **Log** feature in the File menu is a submenu containing three items: File, On and Off. To create a record of a debugging session in the program file, select File to create the new file and turn on the **Log** feature.

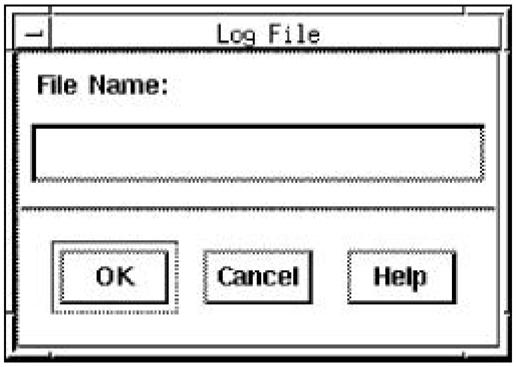


Figure 4-6
Log File dialog

After you have specified the name of a log file, Fx will store a copy of all subsequent commands used into the file. Fx will only log those commands that execute successfully. In addition, you can selectively turn logging on and off with the **On** and **Off** items. You may wish to do this to avoid logging commands, such as the **Print** command, that do not affect program execution.

Using the Read Item

After you have recorded a debugging session, you can use the **Read…** menu item in the File menu to replay it. In addition to recreating entire debugging sessions, you can also use the **Read…** menu item to execute any file of Fx commands. Fx will then execute all of the commands in the file as if you had entered them using the menu items or used the command line box.

When recreating a debugging session, you must make sure that your program is in the same state that is was when you first turned on the **Log** command. For this reason, you may want to make the **Log** menu item the first selection you enter after starting a debugging session.

Chapter Five

Executing Programs in Fx

The previous chapter covered the basics of launching programs in Fx. This chapter covers methods you will use to execute programs within the debugger.

- **Executing Single Statements**

  Discusses the use of the **Step Over** and the **Step Into** buttons to execute single statements.

- **Using Breakpoints**

  Describes what a breakpoint is, how to install and remove breakpoints, and how to automatically execute a list of commands when a breakpoint is encountered.

- **Executing Programs**

  Provides a guide to restarting or resuming program execution during a debugging session.

- **Animating Program Execution**

  Details how to animate program execution by repeating commands.

- **Calling Program Procedures**

  Describes how to test procedures with different arguments and execute special debugging routines.

Executing Single Statements

Instead of executing a program from beginning to end, you may need to execute your program one statement at a time. Fx provides the **Step Over** and **Step Into** buttons to execute single statements. Both buttons are located in the Fx Debugger window.

The Step Into Button

The **Step Into** button will execute the next source line of your program. If the statement that is executed with the **Step Into** button is a call to a subroutine or function, execution will progress through the subroutine or function call and stop at the first executable source line in the function or subroutine.

The Step Over Button

This button also executes single statements. However, it does not follow subroutine or function references, like the **Step Into** button, but rather stops on the next source line of the current procedure.

Returning from Subroutines

When executing your program one statement at a time, you may find that you have pressed the **Step Into** button when you really meant to press the **Step Over** button. Suddenly, your program is stopped in a procedure in which you have no interest. The **Return** button will resume execution until the current procedure returns to its calling point or a breakpoint is encountered.

Using Breakpoints

Although it is possible to use the **Step Over** and **Step Into** buttons to execute your program until you determine where a problem is, this process is inefficient for most programs. Your program may require a complex series of events to occur, or it may need to run for a considerable amount of time before a problem shows up. Using breakpoints allow you to execute your program at full speed until a specific procedure or source line is encountered.

Breakpoints are implemented by replacing the machine language instruction at the specified location with an instruction that will cause your program to stop execution. When this instruction is executed, control returns to Fx, which then determines that a breakpoint has been encountered and restores the original instruction. When execution of the program is then resumed, Fx installs the breakpoint again after executing the replaced instruction. You can set breakpoints from the **Source Pane** popup menu or from the Breakpoint menu.

Using the Source Pane Popup Menu to Set Breakpoints

The **Source Pane** popup menu allows you to set breakpoints at a line of source code or within a subroutine or function in your program.

Setting a Breakpoint on a Source Line

The easiest way to install a breakpoint on a source line is to use the **Toggle Line Break** menu item in the **Source Pane** popup menu.

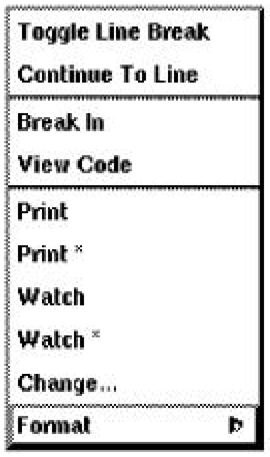


Figure 5-1
Source Pane popup menu

`Example:`

- Triple-click on the source code line in the **Source Pane**.
- Press the third mouse button and choose **Toggle Line Break**.

The installed breakpoint symbol, B, will appear on the left side of the source code line, indicating that the line contains a breakpoint.

Setting a Breakpoint in a Procedure

To set a breakpoint within a function or procedure, use the **Break In** menu item in the **Source Pane** popup menu.

`Example:`

- Select the function or procedure.
- Press the third mouse button and choose **Break In**.

Setting a Breakpoint with the Breakpoint Menu

The **Set…** menu item in the **Breakpoint Menu** allows you to specify breakpoint location, a skip count, and commands to execute when program execution stops at the breakpoint. When this item is selected, the **Set Breakpoint** dialog will appear:

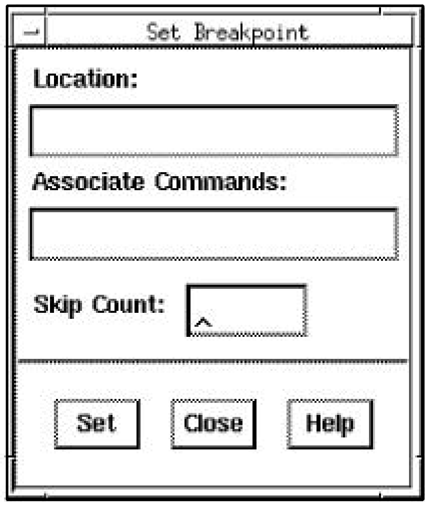


Figure 5-2
Set Breakpoint dialog

Location Text Field

The location of the breakpoint is entered into this field. A breakpoint location can be specified in one of three ways.

1. For source lines enter:

   "filename": line number

or just enter

   :line number

to use the current source file. The line number must be an integer constant. If the line number is not specified, the breakpoint will be set at the first executable line of the file.

2. For procedure names, enter:

   procedure name: line number

If the number 1 is used, the breakpoint will be set on the first line of the procedure. If a line number is not specified with the procedure, the breakpoint will be set on the first instruction of the specified procedure. However, if you stop execution on the first executable instruction, you will not be able to examine or modify the values of local variables and procedure arguments until the procedure's preamble code has been executed.

3. For assembly language addresses enter:

   address

Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant

Associate Text Field

The **Associate** text field names Fx commands to be invoked when a breakpoint is encountered. Multiple commands may be specified by separating each command with a semi-colon. You might use this text field to enter the name of a variable whose value is to be printed each time a breakpoint is reached.

Since you can associate any list of Fx commands with a breakpoint, it is possible to stop program execution, change the value of a variable, and then resume program execution without having to enter the commands each time that the breakpoint is encountered. By doing so, you may be able to temporarily fix a problem without having to edit and compile your source code.

Example:

Consider the following FORTRAN function:

```
      REAL FUNCTION sumarray(array,size)
      REAL array(size),result
      INTEGER i,array_size, size
      array_size = size
      DO 10 i=1,array_size
         result = result+array(i)
10    CONTINUE
      sumarray = result
      RETURN
      END
```

Since the local variable `result` is not initialized to zero, this function will return unpredictable results. This problem can be temporarily fixed by entering

```
sumarray:1
```

into the **Location** text field and the following into the **Associate** text field:

```
change result=0.0;continue
```

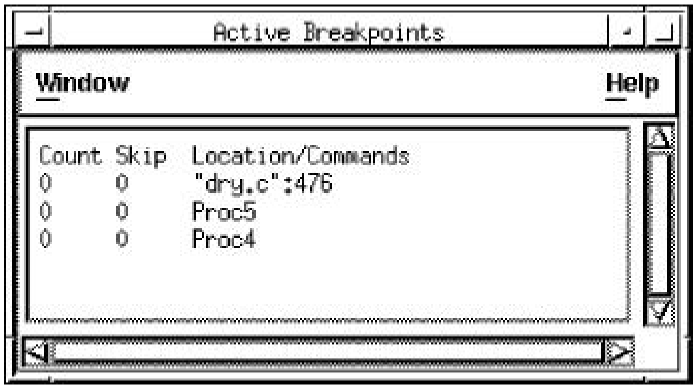
Skip Counts

The **Skip Count** text field allows you to specify the number of times the breakpoint is to be ignored before program execution is stopped. You may find that you have isolated a bug in a specific loop in your program. You can set a breakpoint on the start of the loop and repeatedly press the **Continue** button until you find the problem. However, the problem may only occur on the 100th, 1000th, or 10000th iteration of the loop.

Since pressing the **Continue** button 100 times can be time consuming, Fx provides the **Skip Count** text field as a way to automate this procedure. If you are debugging a loop where an error occurs on the 100th iteration, you can set a breakpoint to be invoked at this error by entering a value of 100 in the text field.

Listing Breakpoints

The **List** item in the Breakpoint menu displays a list of all breakpoints in your program. This list contains each breakpoint location, skip count, number of times the breakpoint has been skipped, and any associated commands.

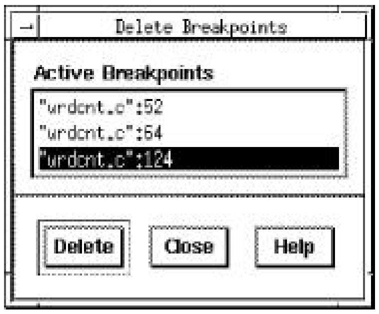


```
Count Skip  Location/Commands
0     0     "dry.c":476
0     0     Proc5
0     0     Proc4
```

Figure 5-3
List Breakpoint window

Removing Breakpoints

You may find that you have set a breakpoint at an inappropriate location, or a breakpoint may have served its purpose and is no longer needed. You can remove one or more breakpoints with the **Delete…** menu item. When you select this menu item, the **Delete Breakpoints** dialog appears.



```
"wrdcnt.c":52
"wrdcnt.c":64
"wrdcnt.c":124
```

Figure 5-4
Delete Breakpoints dialog

To immediately remove all breakpoints from the program, use the **Delete All** menu item.

Executing Programs

Often you may need to specify arguments or resume program execution after debugging with breakpoints. The **Run…** menu item and the **Continue** button allow you to execute programs according to these needs.

Using the Run… Menu Item

The **Run…** menu item restarts program execution, and allows you to specify arguments to the program. When you select the **Run…** menu item from the Execute menu, the **Run Program** dialog appears.

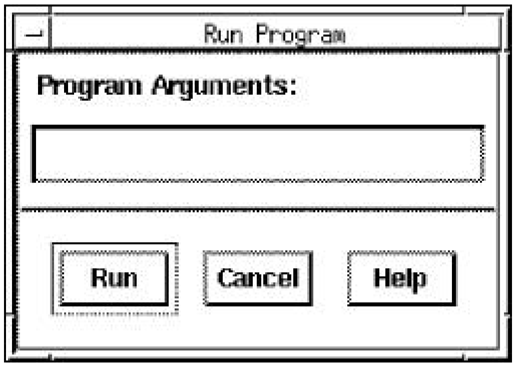


Figure 5-5
Run Program dialog

To restart execution of the current program with the arguments `one two three`:

`Example:`

- Choose the Execute Menu and select the **Run…** menu item.
- In the **Program Arguments** text field, type in `one two three.`
- Press the **OK** button

Once you have specified arguments using the **Run…** menu item, Fx will continue to use those arguments until you specify different ones. If you have specified arguments and then wish to run your program again with no arguments, you can do so by clearing the **Program Arguments** text field or by entering the following in the **Command Entry** text field:

```
change $args = “  “
```

Using the Continue Button

The **Continue** button in the main window resumes execution of a program that has been halted. When invoked, the program is executed until an error occurs, a breakpoint is encountered, or the program runs to completion.

Animating Program Execution

Commands do not have to be issued one at a time. The **Walk…** menu item in the Execute menu provides the ability to repeatedly execute one or more commands.

Start Walk Dialog

When the **Walk…** menu item is selected from the Execute menu, the **Start Walk** dialog appears.

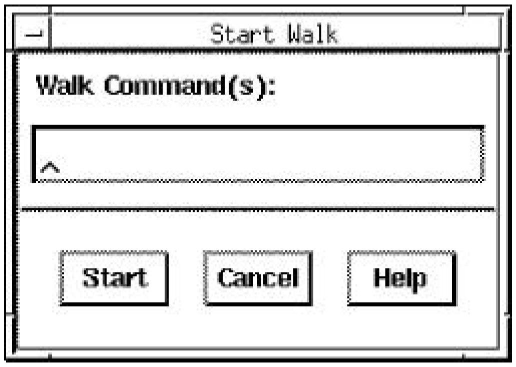


Figure 5-6
Start Walk dialog

For instance, if you wanted to execute the **Step Into** button repeatedly, you would type `step` into the **Walk Command(s)** text field and click on the **Start** button to issue the command.

Stop Walk Dialog

The **Stop Walk** dialog appears after you have initiated the **Walk…** menu item. As the command is repeatedly issued by the **Walk** command, the **Stop Walk** dialog is available to stop the walk process.

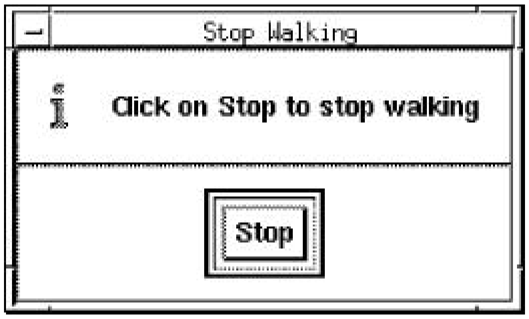


Figure 5-7
Stop Walk dialog

Calling Program Procedures

Any time your program is stopped at a breakpoint, Fx allows you to call any of its functions or subroutines as if they were the next statement to be executed. You can use this feature to test a particular routine with a variety of arguments or to call debugging procedures to format data or test specific conditions.

Using the External Procedure… Menu Item

The **External Procedure** menu item allows you to execute any routine in your program out of sequence. You can pass arguments to the procedure, or set a breakpoint in the procedure and execute Fx commands while this procedure is active.

When a function is called with this menu item, Fx will automatically display the function's result when it returns.

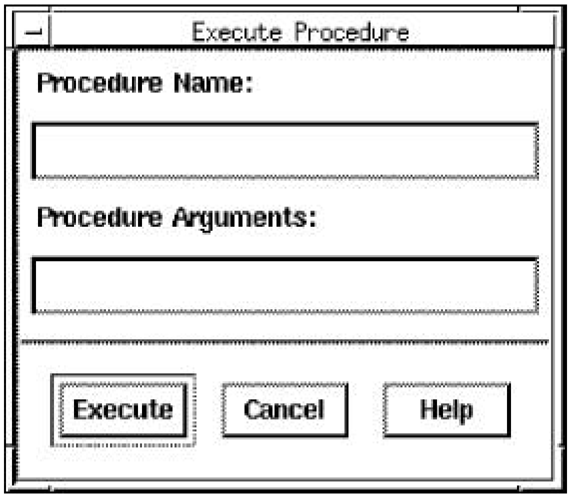


Figure 5-8
Execute Procedure dialog

`Example:`

If you are debugging a FORTRAN program which contains the following function:

```
REAL FUNCTION SUM3REALS(X,Y,Z)
REAL X,Y,Z
SUM3REALS = X+Y+Z
RETURN
END
```

You can execute this function by entering `SUM3REALS` into the **Procedure Name** text field and `1.0,2.0,3.0` into the **Procedure Arguments** text field.

In the preceding example, the arguments to the function `SUM3REALS` were constants, but they can just as easily be variables or expressions. You are not limited to passing constants to routines called with the **External Procedure…** menu item. Keep in mind that Fx does not assure that the number and type of arguments you are passing agrees with what the routine is expecting.

You can also use the **External Procedure…** menu item to format arbitrary portions of program memory as if it represented a particular data structure in your program. In order to do this, you need only create a small debugging function that takes a pointer to the data structure as an argument and returns a pointer to the argument as a result.

Setting Breakpoints in External Procedure

Breakpoints can also be set in a procedure before calling it with the **External Procedure…** menu item. Execution of the procedure will stop just as if it had been called normally. Afterwards, other Fx commands can be used to execute the procedure or to display the procedure's arguments and variables.

Chapter Six

Working with Variables

An advantage of source level debugging is the ability to display the values of program variables without having to insert special debugging statements into your program. This chapter focuses on working with variables during a debugging session.

- **Displaying Variables**

  Presents the menu items for displaying the names, types and memory addresses of your program's variables, arrays and structures.

- **Modifying Variables**

  Provides a guide to changing the values of your program's variables.

- **Finding Overwritten Variables**

  Describes the use of monitors to stop program execution when a variable changes value.

Displaying Program Variables

Viewing program variables can be instrumental in tracking down problems in your code. Fx allows you to examine the contents of your program's variables and data structures whenever your program has stopped execution.

The Print and Print* Menu Items

Showing the value of simple variables is easy; use the **Print** menu item located in the **Source Pane** popup menu. This item displays the contents of your program's variables in the **Output Pane** of the **Fx Debugger** window.

For example, if a FORTRAN program defined the following variables:

```
INTEGER I
REAL X
COMPLEX Z
```

You can display their values by highlighting each variable in the **Source Pane** and choosing **Print** from the **Source Pane** popup menu. The values of `I, X, Z` will be displayed in the **Output Pane**, and will be in the format appropriate for their respective types: `I` as a decimal integer, `X` as a single precision floating point number, and `Z` as a single precision floating point pair. Fx will use the symbol information output by compilers to determine the type and size of a variable.

The **Print*** item is automatically available for source code written in C and dereferences a pointer variable, sending the value to the **Output Pane**.

Displaying Arrays, Structures and Unions

The contents of arrays, structures and unions are displayed in the **Output Pane** with the **Print** menu selection. Individual elements of an array, structure or union are displayed by highlighting the element in the **Output Pane** and choosing the **Print** item from the **Output Pane** popup menu. Pointers to structures and unions can be dereferenced using the **Print*** menu item.

If the name of an array is selected, Fx will display every element using the indexing conventions of the language in which the element was declared. Fx will also use a format appropriate for its type, or an explicit format can be specified with the **Format** submenu.

Setting Formats

The **Format** submenu in the **Source Pane and Output Pane** popup menus allow you to select how variables will be printed in the **Source Pane** and **Output Pane**.

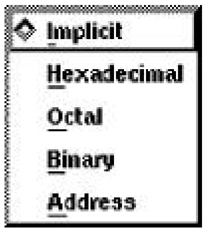


Figure 6-1
Format Submenu

Implicit: The contents of the variable will be printed in the most appropriate format. This is the default setting of the **Format** menu item.

Hexadecimal: Prints the contents of the variable as a hexadecimal integer.

Octal: Prints the contents of the variable as an octal integer.

Binary: Prints the contents of the variable as an binary integer.

Address: Prints the address of the variable.

Evaluating Expressions

In addition to displaying the values of variables, the **Print** selection will also evaluate expressions which involve variables, numeric constants, and source language operators. These expressions can be as simple as adding 10 to the contents of a variable or can include multiple variables and the complete set of FORTRAN intrinsic functions. To evaluate an expression, highlight it and choose the **Print** selection from the **Output Pane** popup menu.

Watch and Watch* Menu Items

At some point during a debugging session, you may want to observe the value of an expression as your program executes. The **Watch** and **Watch*** items are located in the **Source Pane** and **Output Pane** popup menus. Each will invoke a window called **Watch Variables** that shows the value of selected variables as the program executes. The next example shows how to display the value of the array element `a(i,j)` when program execution stops.

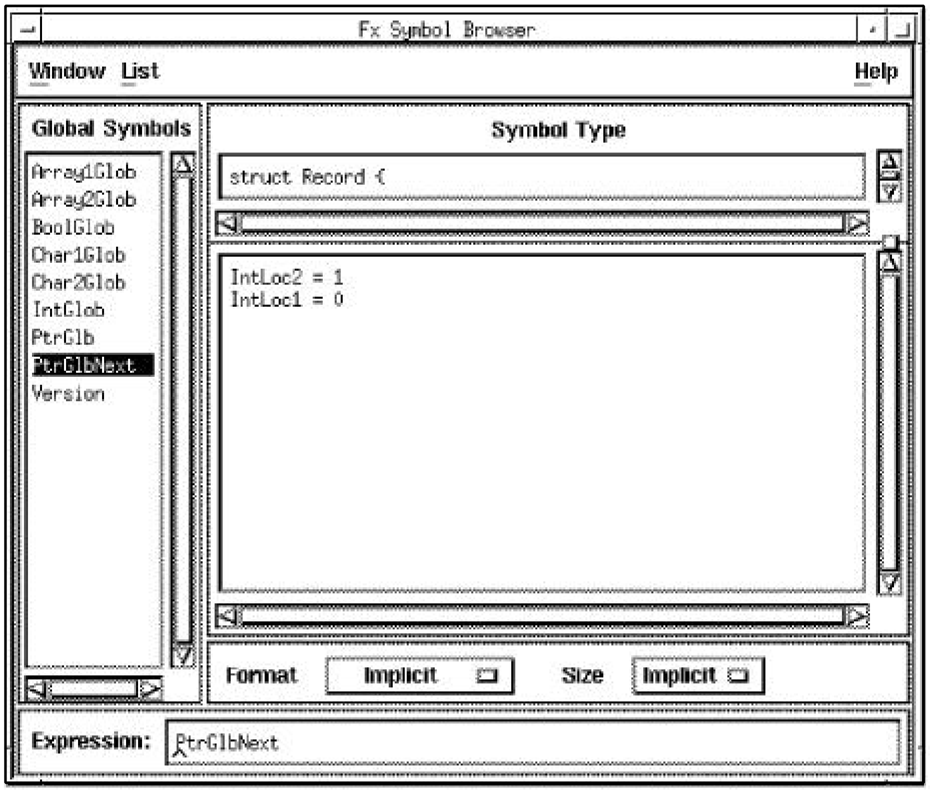
Example:

- Highlight the array element.
- Choose the Watch item to invoke the Watch variables window.
- In the main window, press the **Step Over** button to execute the program. The value of the array element changes in the **Watch Variables** window during execution.

Formatting for the **Watch** and **Watch*** items is set using the **Format** item in the popup menus.

The Symbol Browser

The **Symbol Browser** is used to print and watch the contents of variables, expressions and structures within your programs. The **Symbol Browser** has additional features not available with the **Print** and **Watch** items described above.



```
struct Record {
```

```
IntLoc2 = 1
IntLoc1 = 0
```

Expression: PtrGlbNext

Figure 6-2
Symbol Browser window

Within the **Symbol Browser** window, the menu bar has three menus: Window, List, and Help. The Window menu contains an Erase selection, to clear the contents of the **Output Pane**, and a Close selection to close the window. A list of the variables is shown in the Symbol List on the left side of the window. The List menu allows you to view the different types of variables available: Global, Local, or Static.

Global Symbols

When Global Symbols are selected, the debugger will only list those variables that are declared in the active source file.

Local Symbols

Selecting Local Symbols will display the arguments and local variables for the current subroutine or function.

Static Symbols

Static Symbols item will display the file scope and static variables for the current source files.

When you click an item in the Symbol List, the type of the variable is displayed in the **Symbol Type Pane**. Below it, the **Output Pane** displays the values of expressions. Both the **Symbol Type** and **Output Panes** can be resized using the sash at the top of the lower scroll bar. Both also have a popup menu, like the ones available in the **Fx Debugger** window.

Any expression can be entered in the **Expression** text field. When **Return** is pressed, the result is displayed in the **Symbol Browser Output Pane**.

Setting Formats and Sizes

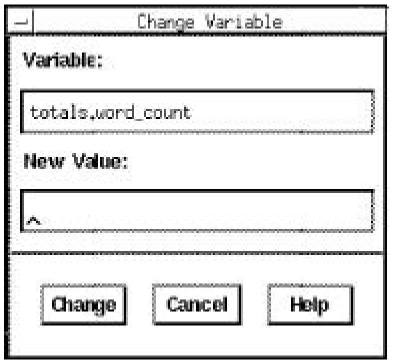
Variables can be printed in additional formats and sizes using the **Format** and **Size** menus. By pressing these menus, a selection can be made from the list of formats and sizes available.

Modifying Variables

Fx allows you to modify the values of program variables when execution of your program is stopped. The **Change…** item is used to assign new values to program variables and Fx control variables.

The Change… Menu Item

The **Change…** menu item is located in both of the popup menus for the main window and the **Symbol Browser**. The dialog allows you the enter the name of the variable and the desired value.



Variable: totals.word_count

New Value:

Figure 6-3
Change Variable dialog

When assigning new values to variables with the **Change…** menu item, Fx will perform appropriate type conversions when possible and inform you when you have specified a value that is inappropriate for the variable you are modifying. New values can be specified as constants or expressions.

Finding Overwritten Variables

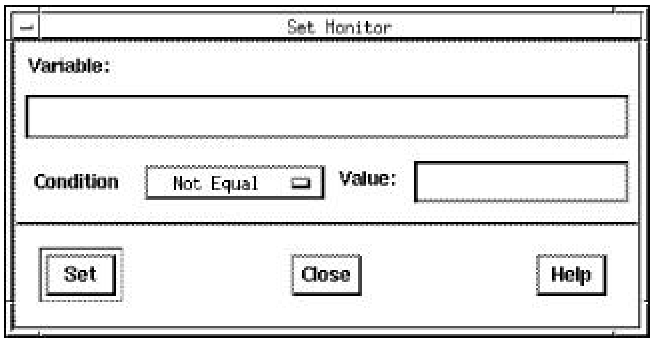
One difficult bug to track down is when a program variable is overwritten by a source line that doesn't appear to reference it. Array boundary errors and invalid pointers are typical causes of this behavior.

Sometimes, you will have no idea where in your program the error occurs. When faced with this type of problem you have little choice but to execute portions of your code and check to see if the variable in question has been modified. This type of debugging is both frustrating and time consuming. Fortunately, Fx provides variable monitors, a feature that will perform much of this task for you.

Using Variable Monitors

When you suspect that the content of a variable is being overwritten, you can use the **Set…** menu item in the **Monitors** menu to stop program execution when the value of that variable changes. You can also install monitors that will halt execution of the program when the value of a variable meets a specific condition.

When you select the **Set…** item, the **Set Monitors** dialog appears and allows you to enter the name of the variable, choose the condition for the monitor value, and enter the value at which to stop program execution. The condition can be set at equal to, not equal, greater than or less than the value.



Condition: Not Equal

Figure 6-4
Set Monitors dialog

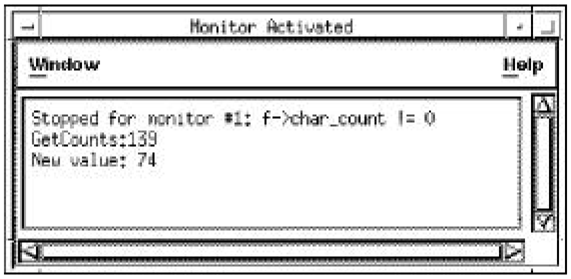
To stop program execution when a variable changes its value:

Example:

- Enter the name of the variable into the **Variable** text field.
- Using the **Condition** menu, choose 'not equal'.
- Click on Set button and click **Close**.

Since the value is not specified, it is implied that the value is equal to the current value.

When the value of the variable is changed, the **Monitor Activated** window will appear and show information on the variable, including its monitor id and current value.



```
Stopped for monitor #1: f->char_count != 0
GetCounts:139
New value: 74
```

Figure 6-5
Monitors Activated window

When the **Continue** button is used with a variable monitor installed, Fx will execute one source statement of your program and then check the value of the monitored variable. If it has changed, execution of the item will be terminated, otherwise the next source statement will be executed. This process is repeated until the monitored variable changes value or the command finishes execution. After you have installed a variable monitor, when you press the **Continue** button, you will notice that execution is much slower.

Removing Monitors

Variable monitors are automatically removed when they stop program execution. However, you may wish to remove one before this occurs. The **Remove**… item in the Monitors menu allows you to highlight and delete monitors.

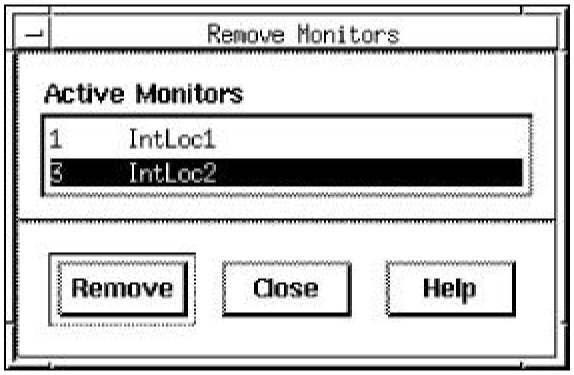
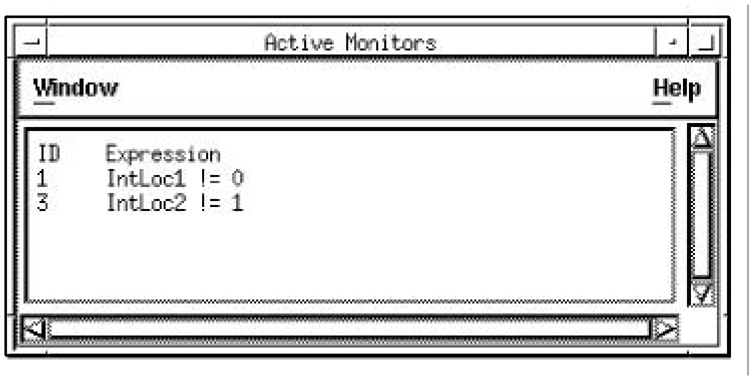


Figure 6-6
Remove Monitors dialog

Listing Monitors

If you want to see the monitors you have installed, you can use the **List**… item to display all of the currently installed monitors. This menu item invokes a dialog which displays the monitor id numbers, all variables being monitored, and the value that the variables had when the monitor was installed. The id of the monitor is useful if you issue the unmonitor command from the **Command Entry** text field.



```
ID    Expression
1     IntLoc1 != 0
3     IntLoc2 != 1
```

Figure 6-7
List Monitors window

Chapter Seven

Assembly Language Debugging

The previous chapters described methods and commands for source level debugging. This chapter describes the features of Fx that support debugging programs at the assembly or machine language level. If most of your programming is done at the source level, you can skip this chapter. However, chances are that eventually you will need to descend to this level of debugging, so you may want to become acquainted with the facilities that are available. This chapter discusses the following topics:

- **Using Fx Commands at the Assembly Level**

  Presents details on using the commands described in the previous chapters when debugging at the assembly language level.

- **Displaying Program Memory and Registers**

  Presents how to display program memory, and how to display and modify the value of program registers.

Using Fx Commands at the Assembly Level

For the most part, the Fx commands presented in Chapters 3, 4 and 5 can be used at the assembly level. This section provides details on using these commands to debug programs at the assembly language level.

Examining Assembly Language Code

When source level debugging information is not available for a procedure, Fx will automatically switch to assembly language debugging. If you are debugging a procedure that has source level information, you can switch to assembly language debugging by using the **Toggle Source** button in the main window.

The **Toggle Source** button switches the **Source Pane** between source and assembly language code. The **Source Pane** will function in the same way in assembly language: text can be viewed using the scroll bar and the commands in the **Source Pane** popup menu are available.

When assembly language is displayed in the **Source Pane**, the **Step Over** and **Step Into** buttons can be used to execute your program one instruction at a time. The **Step Over** button will treat subroutine calls as single instructions.

Resuming Program Execution

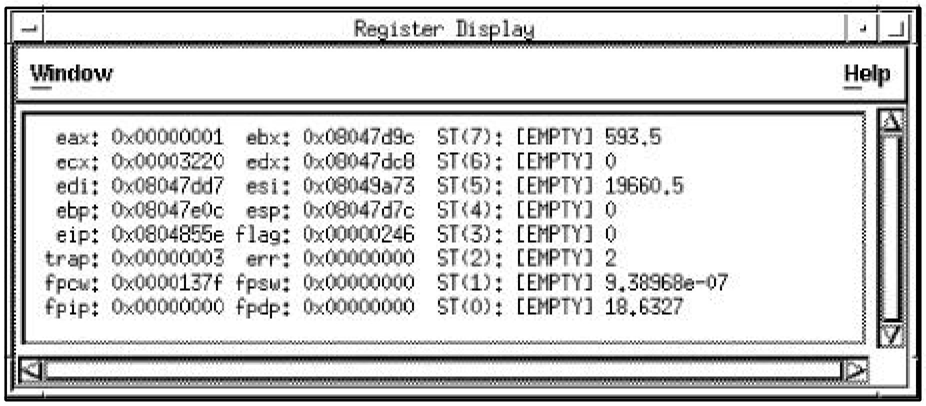
You can use the **Continue** button to resume execution of your program until a particular entry point or instruction is reached. As long as your program has not had its symbol table removed, you can also use the **Return** button to resume execution of your program until the current procedure returns to its calling point.

Setting Breakpoints at the Assembly Level

When debugging at the assembly language level, you can use the Breakpoint menu to install breakpoints on the entry points to procedures and on particular instructions within procedures. You can also specify skip counts for these breakpoints, and use the **Associate** text field to automatically execute commands when a breakpoint is encountered.

Displaying and Changing Registers

You can display and modify the current values of machine registers any time your program has stopped executing. The **Registers** menu item in the Display menu will display the contents of all machine registers.



```
  eax: 0x00000001  ebx: 0x08047d9c  ST(7): [EMPTY] 593.5
  ecx: 0x00003220  edx: 0x08047dc8  ST(6): [EMPTY] 0
  edi: 0x08047dd7  esi: 0x08049a73  ST(5): [EMPTY] 19660.5
  ebp: 0x08047e0c  esp: 0x08047d7c  ST(4): [EMPTY] 0
  eip: 0x0804855e flag: 0x00000246  ST(3): [EMPTY] 0
 trap: 0x00000003  err: 0x00000000  ST(2): [EMPTY] 2
 fpcw: 0x0000137f fpsw: 0x00000000  ST(1): [EMPTY] 9.38968e-07
 fpip: 0x00000000 fpdp: 0x00000000  ST(0): [EMPTY] 18.6327
```

Figure 7-1
Register Display window

If a particular machine provides dedicated floating point registers, the values of these registers will be displayed as floating point numbers. All other registers will be displayed as hexadecimal integers.

Individual registers can be displayed in a variety of formats using the **print** command in the **Command Entry** text field located in the Fx Debugger window. When referring to individual registers, the register name must be preceded by a “$” to distinguish it from symbolic names of variables. For example, to print the value of register `D0`, you would enter `$D0` into the **Command Entry** text field.

Displaying Program Memory

Fx provides a menu selection that displays arbitrary locations in program memory. The **Memory…** menu item in the Display menu allows you to output the contents of a single memory location or a range of memory locations to the **Memory Dump** window. Selecting the **Memory…** menu item invokes the **Display Memory** dialog.

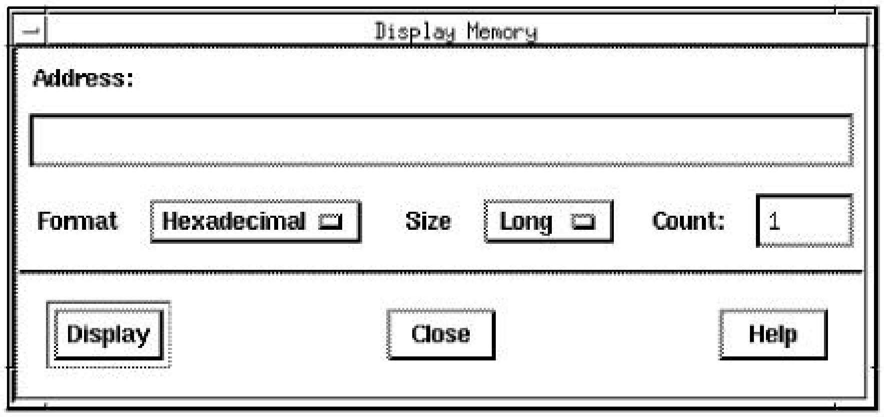
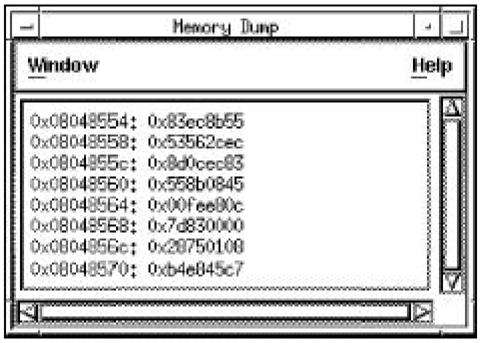


Figure 6-2
Display Memory dialog

The **Address** text field is used to specify the memory address you wish to examine. The **Format** and **Size** option menus allow you to specify the manner in which the contents of the memory address are displayed. The **Count** text field allows you to examine multiple memory locations starting at the specified address.

When the **Display** button is clicked, the memory locations will be shown in the **Dump Memory** window.



```
0x08048554: 0x83ec8b55
0x08048558: 0x53562cec
0x0804855c: 0x8d0cec83
0x08048560: 0x558b0845
0x08048564: 0x00fee80c
0x08048568: 0x7d830000
0x0804856c: 0x20750100
0x08048570: 0xb4e045c7
```

Figure 6-3
Memory Dump window

Monitoring Registers and Memory Locations

You can use the **Set…** item in the Monitors menu to halt execution of the program when the value of a register or memory location changes at the assembly level.

When working with monitors at the assembly level, you will need to use the **Command** submenu in the Monitors menu to change the way Fx executes your program when the **Continue** button is pressed. Select either the **Instruction Step Into** item or the

Instruction Step Over item. When the **Instruction Step Into** item is selected and the **Continue** button is pressed, Fx will execute your program one instruction at a time, checking the value of the monitor after each instruction. When the **Instruction Step Over** item is selected, Fx will treat all subroutine calls as single instructions.

Chapter Eight

Command Arguments

In order get first time users started with Fx as quickly as possible, the previous chapters have glossed over the details of specifying arguments to Fx commands. This chapter describes provides more detail on the items which can be specified as arguments to Fx commands. The chapter covers the following topics:

- **Identifier Scoping**

  Describes the scoping conventions used for Fx command arguments.

- **Specifying Symbols**

  Discusses the interpretation of program variables and procedure names.

- **Specifying Constants**

  Describes the syntax for entering constants as command arguments.

- **Specifying Registers**

  Describes using machine registers as command arguments.

- **Expression Interpretation**

  Discusses the interpretation of variables, entry points and constants when used in expressions.

Identifier Scoping

Identifier scoping refers to the identifiers that are accessible at the current state of the program being debugged. Some arguments are not dependent upon the program and are always available. Program constants, as well as Fx Control Variables, internal variables within the debugger, would be examples of these type of arguments. Control Variables are listed in Appendix B.

Other arguments, such as local variables in the program, are only accessible when the procedure in which they were declared is active. Fx will implicitly determine the appropriate scope, or an identifier's scope can be explicitly stated when necessary.

Implicit Scoping

When identifiers are program specific items, Fx determines the appropriate scope using two sets of scoping information: the *actual scope* and the *current scope*. A program's actual scope is the source line, procedure name and source file which contain the next assembly language instruction to be executed, or the last assembly language instruction executed if a core file is being examined. A program's current scope is the filename, procedure name and source line which appear in the Fx status display.

By default, the current scope is identical to the actual scope. However, the current scope may be changed with the **scope**, **View**, and **view** commands listed in Chapter 9. Fx will use the procedure name defined by the current scope for searching the program's symbol table for local variables, and the file defined by the current scope when searching the program's symbol table for static variables and static functions.

Explicit Scoping

When necessary, it is possible to explicitly specify the scope for local variables. This allows for multiple activations of the same procedure. The syntax for explicitly scoping local variables is as follows:

procedure name {*(level)*}=>*variable name*

where *procedure name* is the name of the procedure where the local variable is declared.

level is an integer constant and specifies a particular instance of the specified procedure in the current chain of program execution. Level zero refers to the most recent instance of the procedure, level one refers to the second most recent instance, and so on. If not specified, *level* defaults to zero.

variable name is the name of the local variable.

For example, if the subroutine `sub` has been called recursively three times resulting in three instances of `sub` in the current chain of execution, the following commands will display the value of the variable `index` for each instance of `sub`.

```
print sub=>(2)index
print sub=>(1)index
print sub=>(0)index
```

Specifying Symbols

This section discuses the interpretation of variable and procedure names.

Symbol Names

The first character of a symbol name must be an upper or lower case letter or an underscore. The remaining characters can be upper or lower case letters, digits, underscores, or dollar signs. A symbol name is terminated by the first occurrence of a character which is not one of the above. Symbol names are significant to 31 characters.

The Fx control variable **$case** controls case sensitivity during symbol table searches. This variable is initially set to "both" causing symbol table searches to be case sensitive. However, it can be set to "lower" or "upper" by entering the **change** command. When **$case** is set to "lower", the symbol name extracted from the command line will be folded to lower case before searching the program's symbol table. When **$case** is set to "upper", the symbol name will be folded to upper case before the search is performed.

On systems where convention dictates the prepending of a special character to symbol names, the Fx control variable **$leading** can be used to eliminate the need to enter this character every time a symbol name is used. For example, if convention dictates that symbol names must have a leading underscore, setting the value of **$leading** to "**_**" will cause Fx to strip a single leading underscore from any name in a program's symbol table, as shown below.

```
change $leading = "_"
```

FORTRAN Symbols
This section describes the FORTRAN data types and symbols understood by Fx and discusses the scoping conventions for each symbol type, the indexing of FORTRAN arrays, and the syntax for specifying character substrings.

FORTRAN Data Types

Fx supports the following FORTRAN data types:

```
INTEGER*1
INTEGER*2
INTEGER
LOGICAL*1
LOGICAL*2
LOGICAL
REAL
DOUBLE PRECISION
EXTENDED
COMPLEX
DOUBLE PRECISION COMPLEX
COMPLEX*24
CHARACTER
RECORD
```

The `INTEGER*1`, `INTEGER*2`, `LOGICAL*1`, and `LOGICAL*2` data types only apply to variables. There is no way to specify an `INTEGER*2` constant. If a constant is assigned to an `INTEGER*2` variable using the **change** command, the constant will be converted before the assignment is performed. The data types `EXTENDED` and `COMPLEX*24` are only supported on machines that perform extended precision arithmetic in hardware.

FORTRAN Subroutines and Functions

FORTRAN subroutine and function names are global to the entire program and are accessible at any time during a debugging session. FORTRAN statement functions are invisible to Fx and cannot be specified as command arguments.

FORTRAN Common Blocks

The names of FORTRAN common blocks are global to the entire program and are accessible any time there is a process or core file active. When used as arguments to commands, the contents of common blocks are assumed to be integers.

FORTRAN Local Variables and Procedure Arguments

The names of FORTRAN local variables and procedure arguments are always local to the procedure or function in which they were declared regardless of their actual location in program memory. Local variables and arguments are accessible when the procedure in which they are declared is defined by the current scope. They may also be explicitly scoped.

FORTRAN Array Indexing

FORTRAN arrays are indexed using the conventions of the FORTRAN language. Indexing is performed in column major order and array indices are specified using standard FORTRAN syntax. Individual array indices may be specified as constants or as expressions involving variables, constants, and operators.

Unsubscripted array names may be specified as arguments to the **print** command causing every element of the array to be displayed. Note that assumed size arrays cannot be displayed in this manner because the size of the last dimension is unknown. The following examples illustrate FORTRAN array indexing.

```
array(1)
array(i,2)
array(i+4,k+3,m)
```

FORTRAN Character Substrings

Substrings of character variables and character array elements may be specified using standard FORTRAN syntax. The substring expressions can be simple integer constants or more complicated expressions involving variables, constants and operators. The following examples illustrate character substring syntax.

```
charvar(1:6)
charvar(i:j)
chararray(1,2)(i+1:7)
```

C Symbols

This section describes the C data types and symbols understood by Fx, and discusses the scoping conventions for each type, C array indexing, dereferencing pointer variables, and referencing members of structures and unions.

C Data Types

Fx supports the following C data types:

```
char
unsigned char
short int
unsigned short int
int
unsigned int
long
unsigned long
float
double
```

Note that many C compilers will not make a distinction between `int` and `long` when producing program symbol information.

C Functions

C function names are global to the entire program unless explicitly declared with the reserved word `static`. Non-static functions are accessible at any time during a debugging session. Static functions are only accessible when the file in which they were declared is defined by the current scope.

C Extern Variables

C variables declared with the reserved word `extern` are accessible any time there is a process or core file. If no type information is available for external variables, the type `int` will be assumed.

C Static Variables

The scoping of variables declared with the reserved word `static` follows the conventions of the C language. If a variable is declared outside of a function, it is only accessible when the file in which it was declared is defined by the current scope. If is it is declared inside a function, it is only accessible when that function is defined by the current scope. Static variables declared inside of functions may also be explicitly scoped.

C Automatic Variables

Automatic variables are only accessible while the function in which they were declared is defined by the current scope. Automatic variables may also be explicitly scoped. Note that Fx does not distinguish between automatic variables declared at the beginning of a function and those declared within a block of the function's statements.

C Array Indexing and Pointer Dereferencing

Array indexing is performed using the conventions of the C language. Indexing is performed in row major order and indices are specified using standard C syntax. Individual indices can be specified as integer constants or as expressions involving variables, constants, and operators.

Unsubscripted array names can be specified as arguments to the **print** command causing every element of the array to be displayed. The following examples illustrate C array indexing

```
array[1]
array[i][1]
array[i+1][j+1]
```

Pointer variables may be dereferenced using the '*' operator, or they may be indexed as if they had been declared as one-dimensional arrays. Consider the following C program fragment:

```
int array[10];
int *aptr;
aptr = array;
```

The following sets of commands will produce equivalent output.

```
print *aptr
print aptr[0]

print *(aptr+8)
print aptr[2]
```

Note that Fx does not multiply the constant 8 by the size of an integer before performing the addition.

C Structure and Union Members

Structure and union members may be specified as command arguments by using the "->" and "." operators. The names of entire structures and unions may be specified as arguments to the **print** command causing every member of the structure or union to be displayed.

Specifying Constants

Constant arguments may be specified in one of the following forms: integer, floating point, complex, or character. The following sections provide details on each of these constant types.

Integer Constants

Integer constants can be entered in decimal, binary, octal, or hexadecimal form.

Decimal Constants

Decimal constants consist of an optional leading sign followed by a string of decimal digits [0-9]. Note that if a sign is not specified and the first digit is a zero, the constant will be interpreted as an octal integer as described below. The following are valid decimal constants:

```
10
-22
+100
```

Binary Constants

Binary constants consist of the letter b or the letter B followed immediately by a string of binary digits [0-1] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits. The following are valid binary constants:

```
b'101'
B'111 111'
```

Octal Constants

Octal constants consist of the letter o or the letter O followed immediately by a string of octal digits [0-7] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits. Octal constants can also be specified using the form familiar to C programmers, where an octal constant consists of a leading digit zero followed by a string of octal digits [0-7]. The following are valid octal constants:

```
o'555'
0555
```

Hexadecimal Constants

Hexadecimal constants consist of the letter z or the letter Z followed immediately by a string of hexadecimal digits [0-9, A-F or a-f] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits. Hexadecimal constants can also be specified using the form familiar to C programmers, where an hexadecimal constant consists of the leading digit zero followed the letter x or the letter X and a string of hexadecimal digits [0-9, A-F or a-f]. The following are valid hexadecimal constants:

```
z'3F'
0x3f
```

Floating Point Constants

A floating point constant consists of an optional sign and string of decimal digits which contains a decimal point. A floating point constant may have an exponent. An exponent is specified by the letter 'E' or the letter 'D' followed by an optional sign and a string of decimal digits. If an exponent character is specified and the fractional portion of the constant is zero, the decimal point may be omitted.

A floating point constant is initially converted to IEEE 96 bit extended precision regardless of the exponent character. After conversion to extended precision, the constant will then be converted to double or single precision depending upon the specified exponent character. Floating point constants specified with a 'D' exponent character will be converted to double precision. Floating point constants specified with an 'E' exponent character, or without an exponent character, will be converted to single precision. The following are valid floating point constants:

```
12.0
-12.999
12.999E12
12.9999D-12
```

Complex Constants

A complex constant consists of a left parenthesis, followed by a pair of floating point constants separated by a comma, followed by a right parenthesis. Double precision complex constants are specified including a 'D' exponent character in one or both of the floating point constants. The following are valid complex constants:

```
(12.0,12.0)
(12.9999E-12,-12.9999E10)
(100.0D0,200.0D0)
```

Character String and C Character Constants

Character string constants are strings of ASCII characters delimited by either apostrophes or quotation marks. The delimiting character may be included in the string itself by representing it with two successive delimiting characters. The following are examples of valid character string constants:

```
"hello world"                 'hello world'
"America's finest"            'America''s finest'
```

If desired, special escape sequences may be embedded in character string constants by using the backslash followed immediately by one of the characters in the following table.

Escape Sequence	Meaning
`\n`	Newline
`\t`	Tab
`\r`	Carriage return
`\f`	Form feed
`\b`	Backspace
`\\`	Backslash
\nnn	octal value specified by *nnn*

The Fx control variable **$escchar** controls whether or not escape sequences are interpreted as described above. When set to zero, which is the default, escape sequences will not be interpreted. **$escchar** must be set to a non zero value using the **change** command before escape sequences will be interpreted.

C character constants are specified as one-byte character strings. The Fx control variable **$escchar** must be set to a non-zero value in order to use escaped characters in C character constants.

Specifying Registers

Registers are entered using the names accepted by the system assembler. In order to distinguish them from symbol names, they must be prefixed with the character "$". When used in expressions, the data type of registers is assumed to be integer. However, if dedicated floating point registers are available they will be typed appropriately. The contents of registers are always retrieved from the actual scope and are available whenever a process or core file is active.

Expression Interpretation

Many Fx commands accept expressions as arguments. Expressions can be simple scalar values, such as a numeric constant or single variable name, or can consist of multiple operands combined with the supported operators for the current expression language.

Current Expression Language

The current expression language is determined by the contents of the Fx control variable **$explang**. By default, this variable is set to "automatic" causing the current expression language to be determined by examining the extension of the file name defined by the current scope. When this file name ends in the characters ".c". the current expression language is C. When the extension is ".f" or ".for", the current expression language is FORTRAN. If desired, the value of this variable may be explicitly set to "C" or "FORTRAN" with the **change** command, allowing expression evaluation in either of these languages regardless of the current scope.

Default Expression Language

When the value of **$explang** is set to "automatic" and it is impossible to determine the appropriate language from the current scope, expressions will be evaluated in the language defined by the Fx control variable **$deflang**. By default, this variable is set to "C" however it may be set to "FORTRAN" using the **change** command.

Supported Language Operators

When specifying expressions as arguments to debugger commands, operands may be combined using the operators of the current expression language. Type conversion between operands and operator precedence follow the conventions of the expression language. Note that parentheses may be used to force a specific order of evaluation regardless of the current expression language.

FORTRAN Operators

The following table lists the supported FORTRAN operators.

FORTRAN operators	
Binary	.NEQV., .EQV., .OR., .AND., .GT.[1], .GE.[1], .NE.[1], .EQ.[1], .LE.[1], .LT.[1], -, +, *, /, **, =
Unary	+, -, .NOT., @[2]

1. The operators .GT., .GE., .NE., .EQ., .LE., and .LT. may also be specified by >, >=, <>, ==, <=, and < respectively.

2. The @ operator is used to specify the "contents of" and can be used to provide an additional level of dereferencing for FORTRAN symbols.

C Operators

The following table lists the supported C operators.

C operators	
Binary	&,|,%,*,+,-,^,.,<,=,>,->,<<,>>,==,!=,&&,||,<=,>=
Unary	~,!,-,&,*,+

FORTRAN Intrinsic Functions

In addition to the language operators listed above, Fx also provides the numeric FORTRAN intrinsic functions for use in expressions. These include the type conversion functions such as FLOAT and INT, the trigonometric functions such as SIN and TAN, and the bit manipulation functions defined by the DOD military standard MIL-STD-1753 such as IOR. The intrinsic functions are available regardless of the current expression language. References to intrinsic functions are distinguished from program entry points with the same name by the presence of an argument list.

Value Expressions

Value expressions evaluate to a single numeric value or character string that can be printed, passed as an argument to an intrinsic function or specified as the value to assign to a variable using the **change** command. Character string expressions are limited to single character string constants, character variables, character array elements, or character substrings. No operators are supported for combining character operands.

When all operands in a value expression are of the same data type, the type of the expression is the same as the type of the operands. When an expression involves operands with different data types, automatic conversion between data types occurs. The data type of the expression result is the data type of the highest operand as defined by the current expression language.

Address Expressions

Address expressions are a subset of possible value expressions and are used to refer to locations in a program's memory space. Since computers are not capable of addressing memory with floating point numbers or character strings, address expressions should only involve integer operands. Although it is possible to specify other types of operands, an error will be reported if the type of an address expression is not integer.

Operand Interpretation

Expression operands are interpreted differently depending upon whether they are used in value expressions or address expressions. The distinction between operand interpretation is generally transparent when debugging programs. Fx is designed to interpret an operand in the manner which makes the most sense for a particular command. For example, when the name of an entry point is used as an argument to the **break** command, Fx will use the address of the specified procedure as the address of the breakpoint.

The interpretation of any argument can be overridden by using source language operators and the FORTRAN intrinsic functions VAL and LOC. Consider the following **print** commands.

```
print main
print LOC(main)
```

The first print command will display the contents of the first memory location for the entry point main. The second will show the address of the entry point main.

The following table lists the basic operands and the ways in which they will be interpreted in value expressions and address expressions.

Operand	In value expressions	In address expressions
constant	numeric value	numeric value
register name	register contents	register contents[1]
variable name	variable contents	variable contents[2]
procedure name	contents procedure's first location when specified alone, procedure address when combined with operators	procedure address
control variable	variable contents	variable address[3]

1. When a register name is used with the **monitor** command, the contents of the specified register name will be monitored for change. This is the only time a register is considered to have an address.

2. The **dump** and **monitor** commands will use the variable address when specified alone and the variable contents when used in expressions.

3. Fx control variables can only appear in address expressions when used with the **change** command.

Chapter Nine

Command Reference

This chapter describes each Fx command. In order to assist in finding a particular command, the commands are presented in alphabetical order and the name of each command is followed by a short description of its purpose.

! *Executing system commands*

Description:

The **!** command suspends a debugging session and executes another command or program. After the command or program finishes, the debugging session is resumed at the point where it was suspended.

Usage:

! *command line*

where *command line* is the name of the command or program followed by its arguments as they would be typed at the shell prompt. All characters following the ! are passed directly to the shell.

Example:

The following command suspends the current debugging session and runs the text editor *vi*(1) on the file `source.f`:

```
! vi source.f
```

Notes:

When entering multiple Fx commands on the same command line, it is permissible to precede the **!** command by other debugger commands.

In order to use the **!** command to recompile the program you are currently debugging, you must use the **kill** command before recompiling. After your program has been compiled, you must use the **reinit** command to make Fx aware of any changes that you made to your program.

Related Commands:

kill, reinit

associate *Executing commands at breakpoints*

Description:

The **associate** command allows debugger commands to be executed when a breakpoint stops program execution. When execution stops at the specified breakpoint, the commands will be executed as if they had been entered from the command line. After the commands have been executed, further commands may be entered.

Usage:

associate *address expression* "*command*"
associate {"*filename*"} {:*line number* } "*command* "
associate {*procedure name* } {:*line number*} "*command* "

where *address expression* evaluates to the address of an instruction in the program. Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

command is any valid debugger command. Multiple commands may be specified by separating them with semi-colons.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the file containing the current source line will be used.

line number is an integer constant. If the specified *line number* does not correspond to an executable source line but is within a procedure in the specified file, the breakpoint will be set on the next executable source line greater than *line number*. If the specified *line number* is between procedures, the breakpoint will be set on the nearest executable source line. If *line number* is not specified, the first executable source line in the specified file will be used.

procedure name is the name of a procedure in the program being debugged. If no *line number* is specified, the breakpoint will be set on the first instruction of the specified procedure. If the specified *line number* is one, the breakpoint will be set on the first executable source line of the specified procedure. If a *line number* other than one is specified, the command is equivalent to "break *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

Abbreviation:

a

Example:

The following command associates a command list with a breakpoint set on the first executable source line of the procedure `subone`:

```
associate subone:1 "print 'At subone'; registers"
```

Notes:

If a breakpoint is currently set at the specified location, the command list will replace any command list currently associated with it. Otherwise, the breakpoint will be set as if it had been specified with the **break** command.

Adding the **continue** command at the end of the list of commands associated with a breakpoint will cause execution to automatically resume after the other commands have been executed.

If an error occurs during execution of a command associated with a breakpoint, any other commands associated that breakpoint will be ignored.

Related Commands:

break, delete, Delete, list breakpoints

break *Setting breakpoints*

Description:

The **break** command is used to place a breakpoint at a given location in the program being debugged. The location can be any valid address expression. The location may also be specified as a file relative source line number or as the first executable source line in a procedure.

Usage:

break {{*address expression*} {,*skip count* }}
break {{"*filename*"} {:*line number* } {,*skip count* }}
break {{*procedure name* } {:*line number*} {,*skip count* }}

where *address expression* evaluates to the address of an instruction in the program. Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

skip count is an integer constant and represents the number of times the breakpoint is to be ignored before program execution is stopped. If *skip count* is not specified, a value of zero will be used causing program execution to stop the first time the breakpoint is encountered.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. If the specified *line number* does not correspond to an executable source line but is within a procedure in the specified file, the breakpoint will be set on the next executable source line greater than *line number*. If the specified *line number* is between procedures, the breakpoint will be set on the nearest executable source line. If *line number* is not specified, the first executable line in the specified file will be used.

procedure name is the name of a procedure in the program being debugged. If no *line number* is specified, the breakpoint will be set on the first instruction of the specified procedure. If the specified *line number* is one, the breakpoint will be set on the first executable source line of the specified procedure. If a *line number* other than one is specified, the command is equivalent to "break *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

Abbreviation:

b

Example:

The following command sets a breakpoint on the location specified by the address expression `main+0x50`:

```
break main+0x50
```

The following command sets a breakpoint on the seventh line of the file `source.f`:

```
break "source.f":7
```

The following command sets a breakpoint on the first executable line of the procedure `subone`. A skip count of two is specified, so program execution will not stop until the third time the breakpoint is encountered.

```
break subone:1,2
```

Notes:

The number of times a breakpoint has been skipped is reset to zero each time program execution is restarted with the **run** command.

Related Commands:

associate, **delete**, **Delete**, **list breakpoints**

change *Modifying variables*

Description:

The **change** command is used to modify the contents of registers, variables, program memory, and Fx control variables.

Usage:

{**change**} *address expression* = *value expression* where *address expression* evaluates to the address of a memory location, a register, or a debugger variable.

value expression specifies the new value to assign to *address expression.*

Abbreviation:

address expression =value expression

Example:

The following command changes the value of the variable `i` to 10:

```
change i=10
```

The following command changes the tenth element of the floating point array `fp_array` to 3.0 :

```
change fp_array(10) = 3.0
```

Notes:

Type coercion will be performed where necessary and possible. For example, it is permissible to assign an integer constant to a floating point variable. However, attempts to assign a floating point constant to a character variable, and vice-versa, will result in an error.

It is not possible to assign values to entire arrays, structures, or unions with the **change** command.

Related Commands:

scope

close *Closing Fx windows*

Description:

The **close** command removes a debugger window from the screen.

Usage:

close {*window name*}

where *window name* is the debugger command which created the window. If *window name* is not specified, the last opened window will be closed. If *window name* specifies a window which is not currently open, nothing will happen.

Abbreviation:

cl

Example:

The following command closes a window created by the **list file** command:

```
close list file
```

The following command closes a window created by the **print** command:

```
close print
```

Notes:

It is not possible to close the standard source window.

Related Commands:

dynamic, **keep**, **size**, **update**

continue *Resuming program execution*

Description:

The **continue** command resumes execution of the program being debugged. If desired, a temporary breakpoint may be specified. When specified, the location of the temporary breakpoint can be any valid address expression, a file relative source line, or the first executable source line of a procedure. Execution continues until the temporary breakpoint is encountered, a breakpoint is encountered, an error occurs, a monitor stops program execution, or the program runs to completion.

Usage:

continue {*address expression*}
continue {{"*filename*"} {:*line number*}}
continue {{*procedure name* } {:*line number*}}

where *address expression* evaluates to the address of an instruction in the program. Examples of useful address expressions include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. If *line number* is not specified, the first executable source line in the specified file will be used. An error will be reported if the specified *line number* does not correspond to an executable line.

procedure name is the name of a procedure in the program being debugged. If no *line number* is specified, execution will continue until the first instruction of the specified procedure. If the specified *line number* is one, execution will continue until the first executable source line of the specified procedure is encountered, otherwise the command is equivalent to "continue *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

If no arguments are specified, execution will continue until a breakpoint is encountered, a monitor evaluates true, an error occurs, or the program runs to completion.

Abbreviation:

c

Example:

The following command resumes execution of the program being debugged until the location specified by the address expression `subone+0x50` is encountered:

```
continue subone+0x50
```

The following command resumes execution of the program being debugged until the first executable source line of procedure `subone` is encountered:

```
continue subone:1
```

Related Commands:

go, **Return**

Delete *Removing all breakpoints*

Description:

The **Delete** command is used to remove all breakpoints from a program. For the description of a command which deletes specific breakpoints, see the **delete** command

Usage:

Delete

Abbreviation:

D

Related Commands

associate, **break**, **delete**, **list breakpoints**

delete *Removing specific breakpoints*

Description:

The **delete** command is used to remove one or more breakpoints from a program. For a description of a command which removes all breakpoints, see the **Delete** command.

Usage:

delete {*breakpoint*}

where *breakpoint* is any of the valid breakpoint specifications. See the **break** command for a description of valid breakpoint specifiers. If *breakpoint* is not specified, the debugger will display each active breakpoint and ask if it should be deleted. If a "Y" or a "y" is entered, the breakpoint will be deleted. If anything else is entered, the breakpoint will be left in place. Note that you do not have enter a carriage return after responding.

Abbreviation:
d

Example:
The following command deletes a breakpoint that was previously set on the first executable source line of the procedure `subone`.

```
delete subone:1
```

Related Commands:
associate, **break**, **Delete**, **list breakpoints**

dump *Displaying program memory*

Description:
The **dump** command displays program memory starting at a specified address.

Usage:
dump *address expression* {#*format* {*repeat count*}}

where *address expression* evaluates to an integer address specifying a location in program memory.

format is any of the legal display formats. Display formats are described in the section on the **print** command. If *format* is not specified, the contents of the specified location will be displayed as a four byte hexadecimal number.

repeat count is an integer constant specifying the number of values to display. When a repeat count greater than one is specified, the specified address will be incremented by the size of the specified format.

Abbreviation:
du

Example:
The following command displays the contents of the memory location `0x402790`:

```
dump 0x402790
```

The following command displays the contents of the memory location specified by the contents of the register `$r1`:

```
dump $r1
```

The following command displays the contents of four consecutive memory locations starting at the address of the variable `index1`:

```
dump index1# 4
```

Related Commands:

change, **print**, **scope**, **section**, **watch**

dynamic *Automating Fx windows*

Description:

The **dynamic** command causes the command which created a specified window to be re-executed after each subsequent debugger command. This command is useful only with the character interface.

Usage:

dynamic {*window name*}

where *window name* is the command which created the window. If *window name* is not specified, the last opened window will be made dynamic. If *window name* specifies a window which is not currently open, nothing will happen.

Abbreviation:

dy

Example:

The following commands will display the register window and cause its contents to be displayed after each subsequent command:

```
register
dynamic register
```

Notes:

The **dynamic** command has no affect when used while debugging with the Fx graphical interface.

Related Commands:

close, **keep**, **shift**, **size**, **watch**

external *Executing procedures out of sequence*

Description:

The **external** command allows program procedures and functions to be executed out of sequence. Arguments may be passed and function results are displayed when a function returns.

Usage:

external *procedure name* {*(arg1,...,argN)*} where *procedure name* is the name of a procedure or function in the program being debugged.

arg1,...,argN are the arguments to the specified procedure. Arguments can be the names of variables accessible from the current scope, constants, or value expressions The debugger does not insure that the specified arguments agree in type or number with what the procedure expects. The debugger will match the calling conventions of the procedure's source language.

Abbreviation:

ex

Example:

The following command executes the function `sum3reals` with the arguments `varone`, 2.0, SIN(1.0):

```
external sum3reals(varone,2.0,SIN(1.0))
```

Assuming that the variable `varone` contains the value 1.0, the debugger will display the return value as follows:

```
External procedure sum3reals returns:                        3.84147
```

Notes:

Use of the **external** command requires that the program being debugged has been linked with the library libg.a or that the operating system supports 88open OCS .tdesc information.

If breakpoints have been installed in the procedure called with the **external** command, execution will stop as if the procedure had been entered through normal program execution. However, only one **external** command can be active at a given time. An attempt to execute a second external procedure before the first one returns will result in an error.

Passing character string constants and entire structures and unions by value is not supported by the **external** command.

Related Commands:

kill, **run**

filestatus *Displaying FORTRAN I/O unit information*

Description:

The **filestatus** command is used to display information about all connected and preconnected FORTRAN units. For units explicitly connected with a FORTRAN OPEN statement, this command displays the unit number, file name, the state of the ACCESS=, FORM=, ACTION=, STATUS= I/O control specifiers used to connect the unit, and the current record number. For preconnected units, this command displays the unit number, and the state of the ACCESS=, FORM=, and ACTION= I/O control specifiers.

Usage:

filestatus

Abbreviation:

f

Notes:

This command will only work for programs which use the Absoft Pro Fortran runtime library.

go *Resuming program execution*

Description:

The **go** command resumes execution of the program being debugged. If desired, a temporary breakpoint may be specified. When specified, the location of the temporary breakpoint can be any valid address expression, a file relative source line, or the first executable source line of a procedure. All breakpoints are ignored during execution of this command. Execution continues until the temporary breakpoint is encountered, an error occurs, a monitor stops program execution, or the program runs to completion.

Usage:

go {*address expression*}
go {{"*filename*"} {:*line* number}}
go {{*procedure name* } {:*line number*}}

where *address expression* evaluates to the address of an instruction in the program. Examples of useful address expressions for the **go** command include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. An error will be reported if the specified *line number* does not correspond to an executable source line. If not specified, the first executable line for the specified file will be used.

procedure name is the name of a procedure in the program being debugged. If no *line number* is specified, execution will continue until the first instruction of the specified procedure. If the specified *line number* is one, execution will resume until the first executable line for the specified procedure is encountered, otherwise the command is equivalent to "go *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

If no arguments are specified, execution will continue until an error occurs, a monitor evaluates true, or the program runs to completion.

Abbreviation:

g

Example:

The following command resumes execution of the program being debugged until the location specified by the address expression `subone+0x50` is encountered :

```
go subone+0x50
```

The following command resumes execution of the program being debugged until the first executable source line of procedure `subone` is encountered:

```
go subone:1
```

Related Commands:

continue, **Return**

help *Getting Help*

Description:

The **help** command is used to access the on line help provided by Fx. When specified with no arguments, a brief introduction to the help system will be displayed including the specific commands and keywords for which help is available.

Usage:

help
help *command*
help *keyword*

Abbreviation:

h

Example:

The following command displays a list of available Fx commands:

```
help commands
```

The following command displays help for the **break** command:

```
help break
```

Instruction *Stepping over assembly language procedures*

Description:

The **Instruction** command executes one or more assembly language instructions, starting with the next instruction to be executed. If one of the instructions to be executed is a call to a procedure, execution of the program will continue until the instruction following the procedure call is encountered or until a breakpoint is encountered in the procedure which is being treated as a single instruction.

Usage:

Instruction {*count* }

where *count* is an integer expression which specifies the number of instructions to execute. If *count* is not specified, one instruction will be executed.

Abbreviation:

I

Example:

The following command executes the next five instructions of the current procedure, treating any procedure calls as single instructions:

```
Instruction 5
```

Notes:

The **Instruction** command only stops for breakpoints set in a procedure which is being treated as a single instruction. Breakpoints set in the current procedure will be ignored.

Related Commands:

instruction, **step**, **Step**

instruction *Executing single instructions*

Description:

The **instruction** command executes one or more assembly language instructions, starting with the next instruction to be executed. If one of the instructions to be executed is a call to a procedure, the procedure will be entered.

Usage:

instruction {*count* }

where *count* is an integer expression which specifies the number of instructions to execute. If *count* is not specified, one instruction will be executed.

Abbreviation:

i

Example:

The following command executes the next five instructions of the current procedure:

```
instruction 5
```

Notes:

The **instruction** command ignores all breakpoints.

Related Commands:

Instruction, **step**, **Step**

keep *Causing Fx windows to remain visible*

Description:

The **keep** command causes the last opened window to remain on the screen during subsequent commands.

Usage:

keep

Abbreviation:

k

Notes:

The **keep** command has no affect when used while debugging with the Fx graphical interface.

Related Commands:

close, **dynamic**, **size**

kill *Terminating the current program*

Description:

The **kill** command kills the current process being debugged without exiting the debugger.

Usage:

kill

Abbreviation:

ki

Notes:

The **kill** command is provided so that a program may be recompiled using the **!** command without leaving the debugger. It is not necessary to use the **kill** command before terminating a debugging session with the **quit** command or when restarting program execution with the **run** command as both of these commands implicitly kill the current process.

Related Commands:

!, **reinit**

list ascii *Displaying ASCII table*

Description:

The **list ascii** command displays a table of printable ASCII characters and their hexadecimal numeric representations.

Usage:

list ascii

Abbreviation:

l a

list breakpoints *Displaying current breakpoints*

Description:

The **list breakpoints** command is used to display all currently active breakpoints, any commands associated with them, their skipcounts, and the number of times they have been encountered without halting program execution.

Usage:

list breakpoints

Abbreviation:

l b

Related Commands:

associate, **break**, **delete**, **Delete**

list control *Displaying Fx control variables*

Description:

The **list control** command displays a list of all currently defined Fx control variables and their current values.

Usage:

list control

Abbreviation:

l c

Notes:

The contents of individual Fx control variables can also be displayed with the **print** command.

Related commands:

change, **print**

list entries *Displaying entry point information*

Description:

The **list entries** command displays the names of selected program entry points, the type of result they return, and their source file and line number. If the source file and line number is not available, the address of the first instruction for an entry point will be displayed.

Usage:

list entries {"*regular expression*"}

where *regular expression* specifies a pattern to match against all the entry point names in the program being debugged. If not specified, all entry point names will be displayed. A complete discussion of regular expressions can be found in *ed*(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

l e

Example:

The following command will list all entry points with names containing the characters `fun`:

```
list entries "fun"
```

The following command will list all entry points with names beginning with the characters `fun`:

```
list entries "^fun"
```

The following command will list all entry points with names ending with the characters `fun`:

```
list entries "fun$"
```

Notes:

The Fx control variable **$elist** determines whether the **list entries** command displays information about all program entry points or only the entry points with full symbol information. By default, this variable is set to zero causing the **list entries** command to skip any entry points without full symbol information. Setting **$elist** to a non-zero value with the **change** command will cause all program entry points to be listed.

The Fx control variable **$lsort** determines whether or not the list of entry points will be sorted. By default, this variable is set to one causing the list to be sorted. For large programs, sorting the list of entry points may take considerable time. Setting **$lsort** to zero with the **change** command will speed execution of this command.

Related Commands:

list globals, **list locals**, **list statics**

list file *Displaying any file*

Description:

The **list file** command allows any file to be examined without leaving the debugger.

Usage:

list file "*filename*"

where *filename* is the name of the file to display.

Abbreviation:

l f

Example:

The following command displays the contents of the include file `source.h`:

```
list file "source.h"
```

Notes:

The **list file** window will remain on the screen until the **close** command is used to remove it.

Related Commands:

close, **next** , **previous**, **search**, **size**, **shift**

list globals *Displaying global symbol information*

Description:

The **list globals** command displays the names of selected global variables for the program being debugged along with their types and locations.

Usage:

list globals {"*regular expression*"}

where *regular expression* specifies a pattern to match against the global variable names in the program being debugged. If not specified, all global variables will be displayed. A complete discussion of regular expressions can be found in *ed*(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

l g

Example:

The following command will list all global variables with names containing the characters `glob`:

```
list globals "glob"
```

The following command will list all global variables with names beginning with the characters `glob`:

```
list globals "^glob"
```

The following command will list all global variables with names ending with the characters `glob`:

```
list globals "glob$"
```

Notes:

The Fx control variable **$glist** determines whether the **list globals** command displays information about all global variables or just the global variables whose data type is known. By default, this variable is set to zero and the **list globals** command will only display global variables with defined data type. To display all global variables, set the value of **$glist** to one with the **change** command.

The Fx control variable **$lsort** determines whether or not the list of global variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For large programs, sorting the list may take a considerable amount of time. Setting **$lsort** to zero with the **change** command will speed execution of this command.

Related Commands:

list entries, **list locals**, **list statics**

list locals *Displaying local variable information*

Description:

The **list locals** command displays the names of selected local variables in the current procedure along with their types and locations.

Usage:

list locals {"*regular expression*"}

where *regular expression* specifies a pattern to match against the local variable names for the current procedure. If not specified, all local variables will be displayed. A complete discussion of regular expressions can be found in *ed*(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

l l

Example:

The following command will list all local variables with names containing the characters `loc`:

```
list locals "loc"
```

The following command will list all local variables with names beginning with the characters `loc`:

```
list locals "^loc"
```

The following command will list all local variables with names ending with the characters `loc`:

```
list locals "loc$"
```

Notes:

The Fx control variable **$lsort** determines whether or not the list of local variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For procedures declaring a large number of local variables, sorting the list may take a considerable amount of time. Setting **$lsort** to zero with the **change** command will speed execution of this command.

Related Commands:

list entries, **list globals**, **list statics**

list monitors *Displaying current monitors*

Description:

The **list monitors** command displays all currently active monitors with their id numbers and monitor expressions.

Usage:

list monitors

Abbreviation:

l m

Related Commands

monitor, unmonitor

list signal *Displaying current signal status*

Description:

The **list signal** command displays the action that will be taken when signals are presented to your program.

Usage:

list signal

Abbreviation:

l si

Related Commands

signal

list statics *Displaying static symbol information*

Description:

The **list statics** command displays the names of selected static variables defined in the current source file.

Usage:

list statics {"*regular expression*"}

where *regular expression* specifies a pattern to match against the static variable names for the current file. If not specified, all static variables will be displayed. A complete discussion of regular expressions can be found in *ed*(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

l s

Example:

The following command will list all static variables with names containing the characters `stat`:

```
list statics "stat"
```

The following command will list all static variables with names beginning with the characters `stat`:

```
list statics "^stat"
```

The following command will list all static variables with names ending with the characters `stat`:

```
list statics "stat$"
```

Notes:

The Fx control variable **$slist** determines whether the **list statics** command displays information about all static variables or just the static variables whose data type is known. By default, this variable is set to zero causing the **list static** command to display only the static variables with defined data types Setting **$slist** to a non-zero value with the **change** command will cause all static variables to be listed.

The Fx control variable **$lsort** determines whether or not the list of static variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For files which declare a large number of static variables, sorting the list may take a considerable amount of time. Setting **$lsort** to zero with the **change** command will speed execution of this command.

Related Commands:

list entries, **list globals**, **list locals**

log *Writing Fx commands to a file*

Description:

The **log** command is used to control logging of debugger commands to a file. Files created by the log command can be played back by using the **read** command.

Usage:

log "*filename* "
log on
log off

where *filename* is the name of the file to store debugger commands in. This form of the **log** implicitly turns on command logging.

on specifies that logging is to resume. If this command is issued without a preceding **log** "*filename*" command, commands will be logged to a file whose name will be constructed using the name of the executable program being debugged with the characters ".fx" appended

off turns off command logging until the next **log** "*filename*" or **log on** command is issued.

Example:

The following command initiates logging of commands to a file named `fxlog`:

```
log "fxlog"
```

Assuming that command logging has not been previously enabled and that the name of the executable program is a.out, the following command initiates logging of commands to a file name `a.out.fx`:

```
log on
```

Notes:

The debugger will only log commands which execute successfully. Commands which contain spelling mistakes or reference symbols incorrectly will not be saved in the log file.

Related Commands:

read

monitor *Stopping execution when a variable changes*

Description:

The **monitor** command is used to install a special form of breakpoint. Unlike traditional breakpoints, which halt execution of a program when a specific instruction is executed, monitors halt execution of a program when a relational expression evaluates true. Monitors can be used to stop execution of a program when a variable changes value, when it equals a specific value, when it is greater than a specific value, or when it is less than a specific value.

Usage:

monitor *address expression* { : *operator value expression* }

where *address expression* evaluates to the address of a location in the program being debugged or a register name. Examples of useful address expressions include the names of local and global variables and subscripted array elements.

operator is one of the supported source language operators for the current expression language.

value expression evaluates to a value to compare against the contents of *address expression* .

If *value expression* and *operator* are not specified, the current contents of *address expression* and the "not equal" operator for the current expression language will be used.

Abbreviation:

m

Example:

The following commands install a monitor on the variable chk_flag and resume execution. Since an operator and value expression are not specified, the monitor will stop execution of the program when the value of `chk_flag` changes:

```
monitor chk_flag
continue
```

As mentioned above, the preceding monitor command is equivalent to

```
monitor chk_flag : .NE. chk_flag
```

when FORTRAN is the current expression language, and

```
monitor chk_flag : != chk_flag
```

when C is the current expression language.

The following command installs a monitor on the variable `index1`. However, an operator and value expression are specified, so this monitor will stop program execution when the value of `index1` is greater than 100:

```
monitor index1 : .GT. 100
```

Notes:

A monitor is automatically removed when it stops execution of the program.

Both *address expression* and *value expression* are evaluated when the monitor is installed. This means that a monitor command such as:

```
monitor array(i,j) : .NE. array(i+1,j+1)
```

will calculate the address of element `array(i,j)` and retrieve the value of element `array(i+1,j+1)` when the monitor is installed. Subsequent changes to the variables `i` ,`j` and array `(i+1,j+1)` will not affect the monitor.

The monitor command does not resume execution of the program. After monitors have been installed, program execution must be resumed with the **continue**, **go**, **step**, or **Step** commands. The **continue** and **go** commands behave differently when monitors are active. Instead of allowing program execution to resume, they repeatedly execute the command defined by the Fx control variable **$mstep**. By default, this variable is set to **step**, causing monitors to be checked after each source statement. This variable can be changed to **Step**, **instruction**, or **Instruction**.

The Fx control variable **$mgrain** determines how many times the command contained in **$mstep** is executed before monitors are checked. The default value for this variable is 1, but it can be set to any positive integer.

The following commands illustrate the use of the Fx control variables **$mstep** and **$mgrain**. The result of these commands is that the value of `index1` will only be checked after every four source statements and any procedure calls will be treated as a single statements:

```
monitor index1
change $mstep = "Step"
change $mgrain = 4
continue
```

Related Commands:

list monitor, unmonitor

next *Scrolling source windows*

Description:

The **next** command is used to scroll the standard source window or **list file** window forward by a specified number of lines.

Usage:

next {*window name* } {,*number of lines* }

where *window name* identifies the window to be scrolled. If not specified, the last opened source window will be scrolled.

number of lines is an integer constant specifying the number of lines to scroll. If *number of lines* is not specified, or zero is specified, the window will be scrolled by the number of lines in the window.

Abbreviation:

n

Example:

The following command scrolls the **list file** window forward six lines:

```
next list file,6
```

Notes:

The **next** command has no affect when used with the Fx graphical interface.

It is only necessary to specify a window name when both the standard source window and the **list file** window are present. Unlike other windows, the standard source window is not created by a specific command. However, the standard source window does have the name **source** associated with it for use with window commands. The following command scrolls the standard source window forward six lines:

```
next source,6
```

Related Commands:

list file, **previous**, **view**, **View**

output *Displaying program output*

Description:

The **output** command is used to display any terminal output produced by the program being debugged.

Usage:

output

Abbreviation:

o

Notes:

The **output** command has no affect when used while debugging with the Fx graphical interface.

This command will generate an error if the **-I** option was used or if output has been redirected to a file using the **run** command.

previous *Scrolling source windows*

Description:

The **previous** command is used to scroll the standard source window or **list file** window backward by a specified number of lines.

Usage:

previous{*window name* } {,*number of lines* }

where *window name* identifies the window to be scrolled If not specified, the last opened source window will be scrolled.

number of lines is an integer constant specifying the number of lines to scroll. If *number of lines* is not specified, or zero is specified, the window will be scrolled by the number of lines in the window.

Abbreviation:

p

Example:

The following command scrolls the **list file** window back six lines:

```
previous list file,6
```

Notes:

The **previous** command has no affect when used with the Fx graphical interface.

It is only necessary to specify a window name when both the standard source window and the **list file** window are present. Unlike other windows, the standard source window is not created by a specific command. However, the standard source window does have the name **source** associated with it for use with window commands. The following command scrolls the standard source window forward six lines:

```
next source,6
```

Related Commands:

list file, **next**, **view**, **View**

print *Displaying program variables*

Description:

The **print** command displays the contents of program variables, registers, Fx control variables, and can also be used to evaluate expressions containing these items as well as constants, source language operators and FORTRAN intrinsic functions. Entire arrays, structures and unions can also be displayed. The **section** command is also available for displaying the contents of arrays.

Usage:

{**print** }*value expression* {# *format*}

where *value expression* specifies the value to be printed. Useful value expressions include variable names, subscripted and unsubscripted array names, structure and union names, and references to structure and union members.

format is one of the valid format specifiers listed in the table below. If *format* is not specified, a format which is most appropriate for the type of *value expression* will be used.

Display Formats:

Display formats are specified by a single character indicating the desired format. Several of the display formats may be followed by a second character which indicates the number of bytes to display in the specified format. The characters used to represent sizes are: **b** (one byte), **s** (two bytes), and **l** (four bytes). The following table lists the display formats which may be specified.

Character	Default Size	Other Sizes	Value displayed as
a	4 bytes	ignored	address of value
b	4 byes	**b,s,l**	binary integer
c	1 byte	ignored	ASCII character
d	4 bytes	**b**,s,**l**	decimal integer
e	8 bytes	ignored	double precision floating point
f	4 bytes	ignored	single precision floating point
o	4 bytes	**b**,s,**l**	octal integer
s	$slen	ignored	null terminated string
u	4 bytes	**b**,s,**l**	unsigned decimal integer
x	4 bytes	**b**,s,**l**	hexadecimal integer

Abbreviation:

value expression

Example:

The following command prints the contents of the variable `varone` in the format most appropriate for its type:

```
print varone
```

The following command prints a single precision representation of PI:

```
print 4.0*ATAN(1.0)
```

The following command prints the variable `intvar` as a hexadecimal integer:

```
print intvar#x
```

Notes:

The only time the **print** must be explicitly specified is when printing the contents of a variable whose name matches one of the debugger commands. For example, a variable named `continue`.

The Fx control variable **$acount** controls the maximum number of array elements to display when an unsubscripted array name is specified. The default value for this variable is 100. However, it can be changed to any positive integer value using the **change** command

The Fx control variable **$union** determines whether or not all members of a union are displayed. By default, this variable is set to one so all members are displayed. Setting **$union** to zero with the **change** command will cause only the first member of a union to be displayed.

The Fx control variable **$slen** controls the maximum number of characters printed when using the **s** format. The default value for this variable is 80 but it can be changed to any positive integer value using the **change** command.

The Fx control variable **$ffmt** controls the display format for single precision values. The default value for this variable is "(1PG15.6E2)". However, it can be changed to any legal FORTRAN format string suitable for writing a single precision value with the **change** command.

The Fx control variable **$efmt** controls the display format for double precision values. The default value for this variable is "(1PG24.15E3)". However, it can be changed to any legal FORTRAN format string suitable for writing a double precision value with the **change** command.

The Fx control variable **$cmpfmt** controls the the display format for FORTRAN COMPLEX values. The default value for this variable is "('(',1PG15.E2,',',1PG15.6E2,')')". However, it can be changed to any legal FORTRAN format string suitable for writing a COMPLEX value with the **change** command.

The Fx control variable **$dcmpfmt** controls the display format for FORTRAN DOUBLE PRECISION COMPLEX values. The default value for this variable is "('(',1PG24.E3,',',1PG24.6E3,')')". However, it can be changed to any legal FORTRAN format string suitable for writing a DOUBLE PRECISION COMPLEX value with the **change** command.

Related Commands:

dump, **scope**, **section**, **watch**

quit

Ending a debugging session

Description:

The **quit** command terminates the current debugging session.

Usage:

quit

Abbreviation:

q

Related Commands:

!, **kill**, **sleep**

read *Reading Fx commands from a file*

Description:

The **read** command allows debugger commands to be read from a file. The file may have been created as the result of using the **log** commands, or can be created by hand using a text editor.

Usage:

read {"*filename*"}

where *filename* is the name of a file which contains valid Fx commands. If *filename* is not specified, a file name consisting of the executable object file with the character ".fx" appended will be used.

If an error occurs while commands are being read from the specified file, the remaining commands in the file will be ignored.

Abbreviation:

rea

Example:

The following command will cause the debugger to execute the commands in the file `fxlog`:

```
read "fxlog"
```

Notes:

It is permissible to nest read commands. That is, a file specified with the **read** command may contain other **read** commands. It should be noted that use of this feature may lead to infinite execution if the file used in a nested **read** command contains a **read** command specifying the original file.

The Fx control variable **$read** can be used to suppress execution of display commands, such as **print** and **registers**, which do not affect the state of the program being debugged. By default, the value of **$read** is set to a non-zero value causing all commands read from a file to be executed. To suppress execution of display commands change the value of **$read** to zero.

Related Commands:

log

registers *Displaying machine registers*

Description:

The **registers** command is used display the current contents of all machine registers.

Usage:

registers

Abbreviation:

r

Notes:

The contents of individual registers can be displayed in a variety of formats using the **print** command.

reinit *Switching programs to debug*

Description:

The **reinit** command is used begin debugging a new program without terminating and restarting Fx.

Usage:

reinit {*executable file* } {-*c corefile* } {-*P pathlist*} {-*p pathlist*}

where *executable file* is the name of a program to begin debugging. If not specified, the name of the last debugged program will be used.

-*c corefile* specifies the name of a core file to use. If not specified and a file named “core” exists in the current directory, this file will be used. To supress the automatic use of a file name “core”, specify -c none. To use a core file named “none”, include a path specification such as -c ./none.

-*P pathlist* and -*p pathlist* specify the directories containing the source files for the new program. The form-*P pathlist* replaces the current set of source paths with the specified pathlist. The form -*p pathlist* adds the specified path list to the current set of source paths.

Example:

The following command will begin a new debugging session on the program `a.out`, supressing any core file, and using the source directories `source1` and `source1`:

```
reinit a.out -c none -P source1:source2
```

Abbreviation:

rei

Related Commands:

!, **kill**

Return *Returning from the current subroutine*

Description:

The **Return** command resumes execution of the program being debugged until the current procedure returns to its calling procedure or a breakpoint is encountered. If the current procedure never returns to its calling procedure, execution will continue until a breakpoint is encountered, an error occurs, or the program runs to completion.

Usage:

Return

Abbreviation:

R

Notes:

If monitors are set during execution of the **Return** command, they will not be checked until the current procedure returns to its calling procedure or a breakpoint is encountered.

Related Commands:

continue, **go**

run *Starting and restarting program execution*

Description:

The **run** command is used to start or restart execution of a program. It can be used to pass arguments to the program and to cause redirection of the program's standard input and output. After the **run** command is issued, the program will execute until a breakpoint is encountered, an error occurs, or the program runs to completion.

Usage:

run {*arg1 arg2 ...argN*}

where *arg1 arg2 ...argN* are the arguments to pass to the program. If an argument begins with the character "<" the remaining characters of that argument, or the following argument if no characters follow the "<", will be used as file name for redirection of standard input. Likewise, if an argument begins with the character ">" the remaining characters, or the following argument if no characters follow the ">", will be used as a file name for redirection of standard output. Redirection of other file descriptors is not supported.

If arguments are not specified, the arguments from the last execution of this command, if any, will be used.

Abbreviation:

ru

Example:

The following command restarts the program being debugged and passes it the character strings `one`, `two`, and `three` as arguments:

```
ru one two three
```

The following command restarts the program being debugged, redirecting standard output to the file `outfile`:

```
r >outfile
```

Notes:

Each time program arguments are specified with the **run** command, the debugger saves a copy of them in the Fx control variable **$args**. If it is desirable to run the program without any arguments, the value of **$args** can be set to a string containing only a single space before issuing the **run** command. The following command will set **$args** to the appropriate value:

```
change $args=" "
```

Related Commands:

external, **kill**

scope *Accessing variables in recursive procedures*

Description:

The **scope** command changes the procedure which defines the current scope. Normally, the current procedure is the procedure whose source or assembly language code appears in the standard source window. However, there may be multiple occurrences of the same procedure in the current chain of execution. The **scope** command provides a method for specifying a particular instance of a procedure.

Usage:

scope *procedure name* {*level*}

where *procedure name* is the name of a procedure present in the current chain of execution. The **trace** command can be used to display a list of all procedure in the current chain of execution.

level is an integer constant and specifies a particular instance of the specified procedure in the current chain of program execution. Level zero refers to the first instance of the procedure, level one refers to the second, and so on. If not specified, the current scope will be changed to the first occurrence of the specified procedure.

Abbreviation:

sc

Example:

The following command changes the current scope to the procedure `main`:

```
scope main
```

The following command changes the current scope to third occurrence of the procedure `subone`:

```
scope subone(2)
```

Notes:

The **scope** command has no effect when registers are displayed.

When debugging C programs, the debugger may not be able to accurately recreate the values of register variables and parameters which passed in registers from different scopes. The Absoft Pro Fortran compilers will automatically store parameters passed in registers in the stack frame when compiling programs for debugging, so the **scope** command will function correctly.

Related Commands:

change, **list locals**, **print**, **section**, **watch**

search *Finding strings in source windows*

Description:

The **search** command searches forward from the current line in the file displayed in the standard source window, or the **list file** window if it is visible, for a line containing a string matching the specified regular expression. A complete description of regular expressions can be found in *ed*(1). If the end of the file is reached without finding a match, the search will resume at the first line of the file and continue until a match is found or the current line is encountered.

Usage:

search {"*regular expression*"}

where *regular expression* specifies the string to search for. If not specified, the last *regular expression* will be used.

Abbreviation:

se

Example:

The following command searches the current source file for a line containing the string `subroutine`:

```
search "subroutine"
```

The following command searches the current source file for a line beginning with the string `1000`:

```
search "^1000"
```

Notes:

The **search** command cannot be used to search disassembled code.

When using the Fx character interface with the **list file** window open, the search will be performed on the file being display in it. When using the Fx graphical interface, the search is always perfomed on the standard source window.

Related Commands:

list file, **view**

section *Displaying arrays*

Description:

The **section** command is used to display specific elements of arrays.

Usage:

section (...(*array name* (*index1,...indexN*),*index1 = lb1,ub1*{,*incr1*})...,*indexN=lbN,ubN*{,*incrN*}) {# *format*}

section *array name* [*lb1*;*ub1*{;*incr1*]...[*lbN*;*ubN*{;*incrN*]{# *format*}

where *array name* is the name of an array accessible within the current scope.

index1...indexN are either symbol names or integer constants used to specify subscript(s) for examining the array. If a given subscript is a symbol name it must have a matching range specification of the form *indexN=lbN,ubN*{,*incrN*}. If a given subscript is a constant, it must not have a matching range specification.

lbN is an integer expression specifying the starting value for *indexN*.

ubN is an integer expression specifying the ending value for *indexN*.

incrN is an integer expression specifying the increment value for a given index. If not specified, a value of one will be used.

format is any of the legal display formats. Display formats are described in the section on the **print** command. If *format* is not specified, a format which is most appropriate for the type of the array will be used.

Abbreviation:

se

Example:

The following commands display every other element of a single dimension FORTRAN array with 20 elements:

```
section (a(i),i=1,20,2)
section a[1;20;2]
```

The following commands display every other element of a single dimension C array with 20 elements:

```
section (a(i),i=0,19,2)
section a[0;19;2]
```

The following commands display elements 1,1 though 20,1 of a two dimensional FORTRAN array:

```
section (a(i,1),i=1,20)
section a[1;20][1;1]
```

The following commands display elements 0,1 though 19,1 of a two dimensional C array:

```
section (a(i,1),i=0,19)
section a[0;19][1;1]
```

Notes:

Symbol names used to define indexes are unique to a **section** command and do not correspond to symbols in the program being debugged. Also, a symbol name used to define an index cannot be used in the integer expressions for *lbN, ubN,* or *incrN.*

The **section** command cannot be used to display arrays which are members of C structures and unions.

Related Commands:

dump, **print**, **scope**, **watch**

shift *Scrolling source windows*

Description:

The **shift** command scrolls the standard source code window and **list file** window horizontally.

Usage:

shift {*direction*}

where *direction* is an integer constant. If *direction* is negative, the windows will be scrolled to the right. If *direction* is positive, they will be scrolled to the left. If *direction* is not specified or zero, the windows will be scrolled as far right as possible.

Abbreviation:

sh

Example:

The following command scrolls the source window one tabstop to the right:

```
shift 1
```

The following command restores the source window to its original position:

```
shift 0
```

Notes:

The **shift** command has no affect when used with the Fx graphical interface.

The Fx control variable **$tabsize** defines the number of spaces for a tabstop. This default value of this variable is eight. However, it can be set to any positive integer value using the **change** command.

Related Commands:

list file, **view**, **View**

signal *Controlling signal actions*

Description:

The **signal** command allows you to control the actions taken when a signal is presented to your program during a debugging session. You can can also use this command to send a signal to your program at anytime.

Usage:

signal *signal number*
signal *signal number* PASS | NOPASS
signal *signal number* STOP | NOSTOP

where *signal number* is the positive integer which represents the signal you wish to control. Specifying only a signal number will cause that signal to be presented to your program the next time execution is resumed. You can remove any pending signal by specifying zero instead of a signal number.

PASS | NOPASS indicates whether or not your program should be allowed to see a particular signal. Specify PASS if you want the signal to be passed on to your program or NOPASS if you wish Fx to prevent your program from receiving the signal.

STOP | NOSTOP indicates whether or not a signal should stop execution of your program. Specify STOP if you want Fx to stop execution of your program when the signal occurs and NOSTOP if your program should be allowed to continue executing.

Abbreviation:
sig

Examples:
The following command will cause your program to receive a floating point exception signal the next time execution is resumed:

```
signal 8
```

The following command will prevent your program from seeing future occurrences of the floating point exception signal:

```
signal 8 NOPASS
```

The following command will prevent Fx from stopping execution of your your program when a floating point exception signal occurs:

```
signal 8 NOSTOP
```

Notes:
The **list signal** command displays the current signal settings.

Related Commands:
list signal

size *Resizing Fx windows*

Description:
The **size** command changes the size of a visible debugger window.

Usage:
size {*window name* } {, *number of lines* }

where *window name* specifies the desired window. If *window name* is not specified, the size of the last opened window will be changed.

number of lines is the integer number of lines by which to increase or decrease the window's size. If positive, the window's size will increase. If *number of lines* is not specified, the value one will be used.

Abbreviation:
si

Example:

The following command increases the size of the last opened window by three lines:

```
size ,3
```

Notes:

The **size** command has no affect when used while debugging with the Fx graphical interface.

The standard source window is not affected by the **size** command.

Related Commands:

close, **dynamic**, **shift**

sleep *Delaying command execution*

Description:

The **sleep** command suspends execution of the debugger for a specified number of seconds.

Usage:

sleep {*number of seconds*}

where *number of seconds* is a positive integer constant. If not specified or zero is specified, a value of one will be used.

Abbreviation:

sl

Notes:

The **sleep** command exists primarily for creating self running demonstrations of the debugger. However, there may be occasions to use it for other purposes when using the Fx character interface. Consider the following **associate** command:

```
associate main:1 "print i;print j;continue"
```

Since there are two print commands, the value of `i` may be replaced by the value of `j` before there is time to examine it. The following **associate** command will insert a delay between the two print commands show that both values may be examined:

```
associate main:1 "print i;sleep 5;print j;continue"
```

Step *Stepping over procedure calls*

Description:

The **Step** command executes one or more source statements, starting with the next statement to be executed. If one of the statements to be executed is a call to a procedure, execution of the program will continue until the statement following the procedure call is encountered or a breakpoint is encountered.

Usage:

Step {*count* }

where *count* is an integer expression which specifies the number of statements to execute. If *count* is not specified, one statement will be executed.

Abbreviation:

S

Example:

The following command executes the next five statements of the current procedure, treating any procedure calls as single statements:

```
Step 5
```

Notes:

The **Step** command will only stop for breakpoints set in a procedure which is being treated as a single statement. It will ignore any breakpoints set on the instructions which are part of the source statement being executed.

Related Commands:

instruction, **Instruction**, **step**

step *Executing single source statements*

Description:

The **step** command executes one or more source statements, starting with the next statement to be executed. If one of the statements to be executed is a call to a procedure, the procedure will be entered if complete symbol information is available for it. If complete symbol information is not available, execution of the program will continue until the statement following the procedure call is encountered.

Usage:

step {*count* }

where *count* is an integer expression which specifies the number of statements to execute. If *count* is not specified, one statement will be executed

Abbreviation:

s

Example:

The following command executes the next five statements of the current procedure:

```
step 5
```

Notes:

The **step** command will ignore any breakpoints set on instructions which are part of the source statement being executed.

Related Commands:

instruction, **Instruction**, **Step**

trace *Displaying a stack trace*

Description:

The **trace** command is used to display the chain of procedure calls which produced the current program state. Each procedure in the current chain is listed along with the procedure that called it and the calling procedure's file and line number if available.

Usage:

trace

Abbreviation:

t

Notes:

This command may not function correctly if the program being debugged is stopped during execution of the entry code for a procedure. When the complete entry code of a the procedure has not yet been executed, the words "At Entry" will appear in the status display.

unmonitor *Removing monitors*

Description:

The **unmonitor** command removes a monitor previously installed with the **monitor** command.

Usage:

unmonitor {*monitor id*}

where *monitor id* is the positive integer value associated with the monitor when it was installed by the **monitor** command. The **list monitors** command can be used to determine what id is associated with a particular monitor.

If *monitor id* is not specified, the debugger will display each installed monitor and ask if it should be removed. If a "Y" or a "y" is entered, the monitor will be removed. If anything else is entered, the monitor will remain active. Note that you do not have enter a carriage return after responding.

Abbreviation:

un

Example:

The following command removes the monitor which was assigned id 3 by the **monitor** command:

```
unmonitor 3
```

Related Commands:

list monitors, **monitor**

unwatch *Removing watch variables*

Description:

The **unwatch** command allows you to remove individual watch variables.

Usage:

unwatch {*watch id*}

where *watch id* is the positive integer value associated with the watch variable when it was created with the **watch** command. Watch ids are displayed in parentheses before each watch variable.

Abbreviation:

unw

Example:

The following command removes the watch variable with id 1:

```
unwatch 1
```

Notes:

Removing the last watch variable with the **unwatch** command automatically closes the watch variable window.

Related Commands:

watch

update *Refreshing Fx display*

Description:

The **update** command redraws the contents of all visible windows. This command is useful if the windows have become disrupted.

Usage:

update

Abbreviation:

u

Notes:

The **update** command has no affect when used with the Fx graphical interface.

View *Displaying assembly language code*

Description:

The **View** command is used to examine the assembly language code for the program being debugged.

Usage:

View {*address expression*}
View {{"*filename*"} {: *line number*}}
View {{*procedure name*} {: *line number*}}

where *address expression* evaluates to the address of an instruction in the program. Examples of useful address expressions for the **View** command include the name of an entry point, the name of an entry point plus an integer offset, or the contents of a pointer to a function.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the name of the current source file will be used.

line number is an integer constant. If the specified line number exceeds the number of actual lines in the source file, assembly language code for the last executable line of the source file will be displayed.

procedure name is the name of a procedure in the program being debugged. If the specified *line number* is one, the assembly language code for the first executable source line of the specified procedure will be displayed. If a *line number* other than one is specified, the command is equivalent to "View *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

If no arguments are specified, the address of the next instruction to be executed will be used.

Abbreviation:

V

Example:

The following command displays assembly language code starting at the address `main+0x50`:

```
View main+0x50
```

The following command displays assembly language code for the procedure `subone`:

```
View subone
```

Notes:

The **View** command implicitly changes the current scope to the file and procedure being displayed. If the procedure which is not currently active, the words "Not Active" will appear in the status display.

Related Commands:

list file, **next**, **previous**, **view**

view *Displaying source code*

Description:

The **view** command is used to examine the source code for the program being debugged. For details on a command which allows any file to be examined see the **list file** command.

Usage:

view {*address expression*}
view {{"*filename*"} {:*line number* }}
view {{*procedure name* }{:*line number*}}

where *address expression* evaluates to the address of an instruction in the program. Examples of useful address expressions for the **view** command include the name of an entry point, the name of an entry point plus an integer offset, or the contents of a pointer to a function.

filename is the name of a source file which is part of the program being debugged. If *filename* is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. If the specified line number is greater than the number of actual lines in the source file, the last line of the source file will be displayed.

procedure name is the name of a procedure in the program being debugged. If the specified *line number* is one, the first executable source line of the specified procedure will be displayed. If a *line number* other than one is specified, the command is equivalent to "view *filename* :*line number*", where *filename* is the name of the file which contains the specified procedure.

If no arguments are specified, the line number of the next statement to be executed and the source file which contains it will be used.

If source line information is not available for a specified procedure or if the specified address expression does not identify a line of source code, assembly language will be displayed instead.

Abbreviation:

v

Example:

The following command displays the source code for the procedure `subone`:

```
view subone
```

The following command displays the source code for the file `source.c`, starting with line 100:

```
view "source.c":100
```

Notes:

The **view** command implicitly changes the current scope to the file and procedure being displayed. If the procedure is not currently active, the words "Not Active" will appear in the status display.

Related Commands:

list file, **next**, **previous**, **search**, **View**

walk *Repeatedly executing Fx commands*

Description:

The **walk** command allows other debugger commands to be repeatedly issued by the debugger itself. After the **walk** command has been issued, the debugger will continue to execute the commands contained in the debugger variable **$walkcmds** until explicitly stopped, an error occurs, a breakpoint is encountered, a monitor stops execution, or the program runs to completion.

Usage:

walk {*speed*}

where *speed* is an integer constant between one and ten which specifies the delay in half-seconds between issuing commands. If *speed* is not specified no delay will occur between commands.

Abbreviation:

w

Notes:

By default, **$walkcmds** contains the character string "step". This can be changed to any character string using the **change** command. Setting this variable to a nonsense string will result in error and terminate execution of a **walk** command immediately.

watch *Observing program variables*

Description:

The **watch** command is used to automatically display the value of an expression after each subsequent debugger command.

Usage:

watch *value expression* {# *format* }

where *value expression* specifies the value to be printed. Particularly useful value expressions include variable names, subscripted arrays, and references to structure and union members.

format is any of the legal display formats. Display formats are described in the section on the **print** command. If *format* is not specified, a format that is most appropriate for the type of value expression will be used.

Abbreviation:

wat

Example:

The following command will cause the contents of the variable `index1` to be displayed after each subsequent command:

```
watch index1
```

Notes:

Value expressions involving local variables will only be displayed while the procedure in which they are declared is active. When a procedure, which declares a variable that is part of watched expression, is inactive, the message "(OUT OF SCOPE)" will be displayed instead of the expression's value.

Related Commands:

dump, **print**, **scope**, **section**

APPENDIX A

The Fx Character Interface

The Fx character interface provides full screen debugging capabilities on a wide variety of character based terminals. While it cannot provide all of the benefits of a graphic user interface, you will find that it does offer significant advantages over line oriented debuggers.

Starting the Character Interface

When using the Fx character interface you must make sure that Fx can determine the type of your terminal. Fx uses the environment variable `TERM` to find out this information. If you are a regular user of a full screen editor, such as *vi*(1), this variable is already set to the correct value. If this variable is not defined, you will see the following error message when you begin a debugging session:

```
Unable to initialize Fx interface, check the value of TERM.
```

If this occurs, you must determine the name of your terminal and define `TERM` appropriately. If you do not know how to define your terminal, consult your system's User Guide or ask your system administrator to assist you.

Like many programs, Fx will supply default arguments where none are specified. For example, the standard UNIX C compiler, as well as many other compilers, will create an executable program named `a.out` unless you tell it otherwise. Similarly, Fx will assume you wish to debug a program named `a.out` unless you tell it otherwise. For example, if you have a source file named `wrdcnt.c,` you can compile it with the following command:

```
cc wrdcnt.c
```

The C compiler will look in the current directory for a file named `wrdcnt.c` and produce an executable program named `a.out`. To debug this program, you can enter:

```
fx
```

Fx will look in the current directory for a program named `a.out` and begin a debugging session.

Preparing Your Program For Debugging

If you performed the steps described in the previous section, you probably noticed that Fx displayed a screen full of assembly language code after issuing the following warning message:

```
Warning: a.out has no line number information
```

This occurred because you did not inform the C compiler that you intended to use a source level debugger. In order to debug at the source level, you must specify the **-g** option when compiling your source code. By specifying the **-g** option, you instruct compilers to include additional information in the executable program that describes the source files, procedures, and variables that are part of your program.

You can use the following command to compile the source file `wrdcnt.c` for use with a source level debugger:

```
cc -g wrdcnt.c
```

The C compiler will produce a program named `a.out` and you can start a debugging session by entering:

```
fx
```

Instead of issuing a warning message, Fx will display the following:

```
Loading symbol information
Processing wrdcnt...
Symbol table initialization complete
Enter Cr to continue
```

After you entered a carriage return, Fx will display the source code for `wrdcnt.c` and you can begin debugging.

Starting an Fx Debugging Session

This section discusses starting a debugging session. It describes how to specify the name of the program to debug, the name of a core file, and the directories that contain a program's source code. It also discusses the command line options that you can use to modify the behavior of Fx.

Command Line Syntax

Fx is invoked from the shell prompt as follows:

`fx {`*`program name`*`} {`*`options`*`}`

where `fx` starts a debugging session using the character interface.

program name is the name of the program to be debugged. If no name is specified, the name a.out will be used.

options is a list of options in the form `-o1 -o2` ... where `-o1`, `-o2` ... represent any of the command line options discussed in below.

Sample Command Lines

This section contains sample command lines that might be used to invoke Fx. Each example is followed by a brief discussion of how the command line arguments are interpreted.

The following command line is the simplest way to invoke Fx:

```
fx
```

When no arguments are specified, Fx assumes the following default arguments:

```
fx ./a.out -c ./core -p ./
```

In this case, the program, the core file, and any source files used to build the program are assumed to be the current working directory. The following command line illustrates how to specify a program name, core file, and source paths.

```
fx myprogram -c lastcore -p interface:data:sysdep
```

The `-c lastcore` causes Fx to look in the current directory for a file named `lastcore` to use with `myprogram`. The string `interface:data:sysdep` indicates that the source files which were used to make `myprogram` exist in three subdirectories off of the current working directory.

Warnings and Progress Information

When you start a debugging session, Fx reads the debugging information contained in your program and informs you of any unusual conditions such a source file that has changed since you compiled your program. If you wish to suppress the printing of this information, specify the **`-w`** option when you begin a debugging session.

Specifying a Core File

If a file named `core` exists in the same directory as the program being debugged, Fx will automatically use it. If you wish to use a core file with a different name or location, you must specify the **`-c`** *name* option when starting Fx. If you wish to suppress the automatic use of a file named `core`, use the **`-c`** option with the name `none`. Note that if you must use a core file named `none`, you will have specify a partial path name like `./none`. You can enter the following command to debug a program named `program1` with the core file `/usr/fred/core`:

```
fx program1 -c /usr/fred/core
```

Specifying Multiple Source Directories

When you specify the **-g** option, compilers will cause the name of each source file to be present in the executable program. However, the directory where the source file is located is not present. If your program is built from source files contained in multiple directories, you will need to specify those directories using the **-p** *pathlist* option. You may specify all of the directories in one path list by separating them with colons or you may specify the **-p** option multiple times.

To start a debugging session on a program named `program1` created with source code located in the directories `/usr/fred/source` and `/usr/fred/interface`, you can enter either:

```
      fx program1 -p /usr/fred/source -p /usr/fred/interface
```
or:
```
      fx program1 -p /usr/fred/source:/usr/fred/interface
```

Interception of terminal input/output

When you are using the Fx character interface, Fx will intercept any terminal I/O performed by your program. If you are debugging a program that makes minimal use of terminal I/O, you may wish to disable this feature by specifying the **-I** option. If you do, you should note that any terminal I/O your program does will overwrite the Fx display. When this occurs, you can use the **update** command to redraw the screen. Note that this option has no effect when used with the Fx graphic interface.

Interception of FORTRAN I/O gotos

When debugging FORTRAN programs compiled by an Absoft Pro Fortran compiler, Fx will intercept execution of a FORTRAN ERR= or END= I/O specifiers and report that program control was transferred abnormally. If you do not want this to occur, specify the **-G** option. If you are debugging a C program or a FORTRAN program compiled with a different compiler this option has no effect.

Debugging Curses Applications

If you are debugging an application that uses the curses library with the Fx character interface, you will need to specify the **-C** option to prevent conflicts between the Fx curses interface and your program. Note that this option implicitly turns on the **-I** option. When the Fx display is disrupted by the actions of your program, you can use the **update** command to correct this problem.

Altering Option Defaults

If you find that you are frequently specifying one or more of the command line options, you can reverse the default state of the command line options. Before interpreting options on the command line, Fx looks for an environment variable named `FXOPTIONS`. If this variable exists, its contents will be used to set the default option states.

Character Interface Tutorial

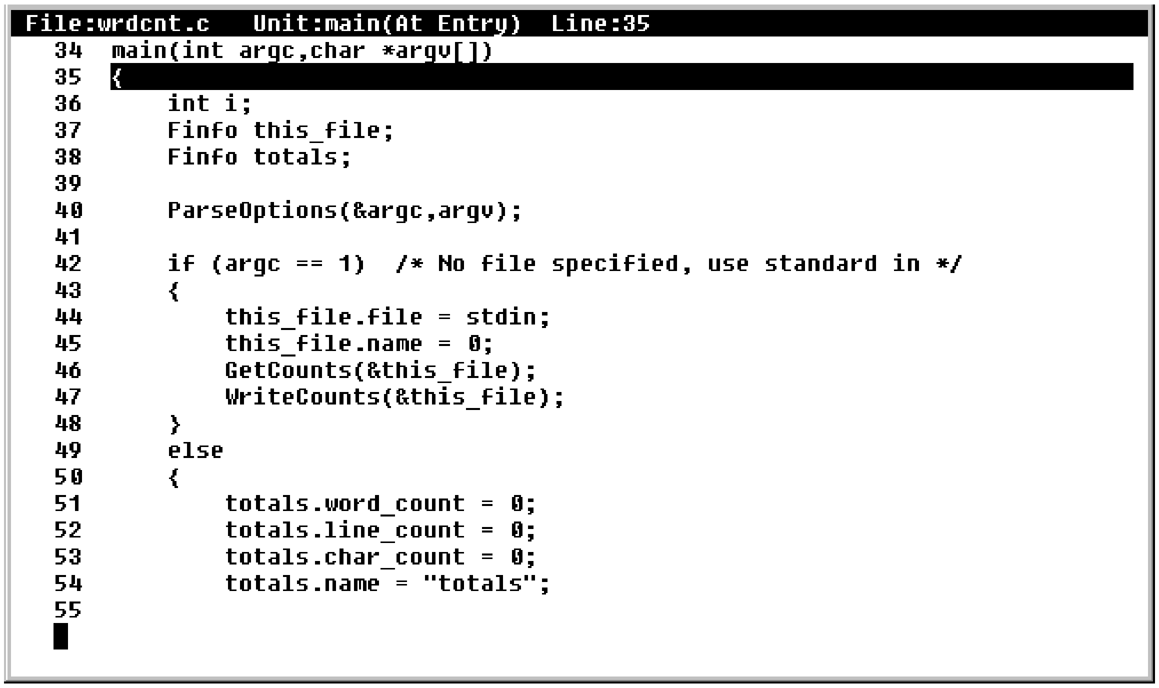
To load the tutorial program, type:

```
fx wrdcnt
```

and press Return. Information on the symbol table and copyright will appear on the screen. To continue, press the Return key.

Character Interface Windows

The Fx character interface divides your terminal screen into separate windows. There are three windows that are normally visible during a debugging session: the **Command** window, the **Status** window, and the **Source** window. Additional display windows are created when you execute a command, such a the **print** command, which displays information about your program.



```
File:wrdcnt.c   Unit:main(At Entry)  Line:35
main(int argc,char *argv[])
{
    int i;
    Finfo this_file;
    Finfo totals;

    ParseOptions(&argc,argv);

    if (argc == 1)  /* No file specified, use standard in */
    {
        this_file.file = stdin;
        this_file.name = 0;
        GetCounts(&this_file);
        WriteCounts(&this_file);
    }
    else
    {
        totals.word_count = 0;
        totals.line_count = 0;
        totals.char_count = 0;
        totals.name = "totals";

```

Figure A-1
Character interface

The Source Window

The **Source** window occupies all of the lines between the status window and the command window. This window is used to display the source and assembly code for your program. When a line on which a breakpoint has been set is visible in the source code window, the character 'B' will be displayed next to the line. You can use the **next** and **previous** commands to scroll the source window vertically, and the **shift** command to scroll the source window horizontally.

The Command Window

You communicate with Fx by typing commands on the **Command** window at the bottom of the screen. This window occupies the last line on the screen, and its location is marked by the command prompt.

After you have typed a command, you can have Fx execute the command by pressing the return or enter key. Multiple commands may be specified by separating them with semi-colons. You can repeat execution of the last command(s) without retyping the command by pressing the Return or Enter key.

Although most character terminals cannot display more than 80 characters on a single line, you can enter up to 256 characters on the command line. If you enter more than 80 characters the command line will scroll to the left so that you can continue typing. By default, the command prompt is "Fx>" but this default can be modified using the Fx control variable **$prompt**. Control variables are discussed in Appendix B.

The Status Window

The first line of the screen is reserved for the **Status** window. This window is used to display information about the current state of your debugging session. Information displayed in this window includes the current source file, procedure and line number as well as any Fx features, such as command logging, which are currently active.

Display Windows

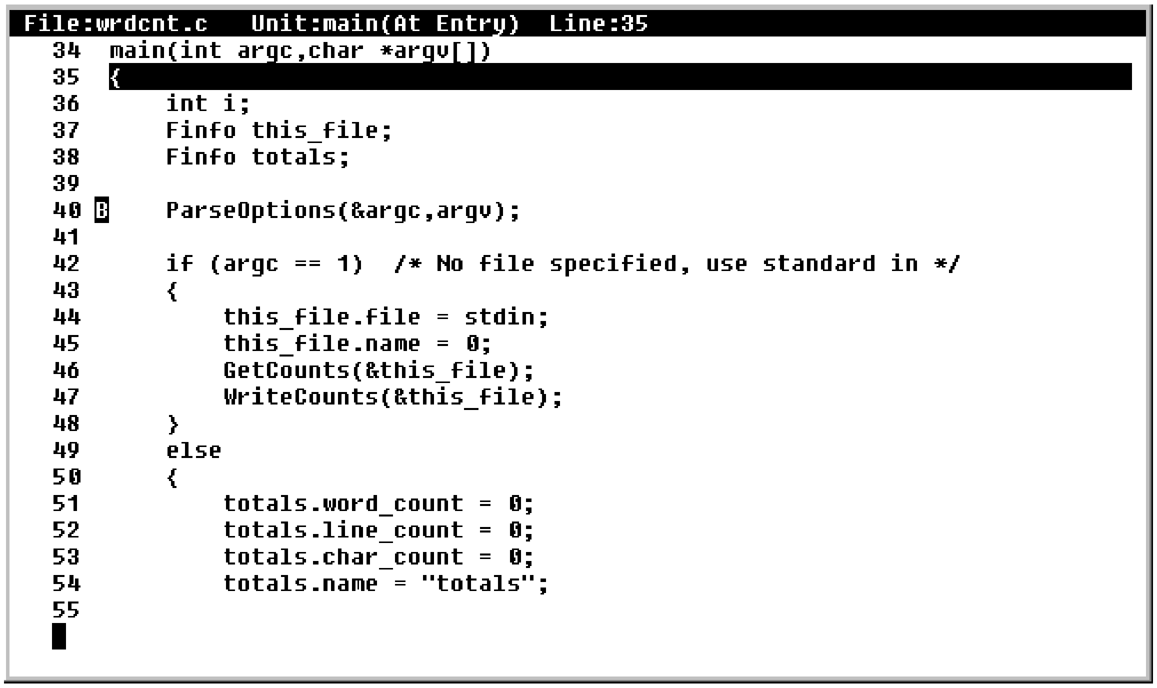
When you execute a command that displays information about your program, a **Display** window is created to contain the information. These windows appear above the command window and reduce the size of the source window. Fx will attempt to display all of the output of a command by increasing the size of a display window. Each display window includes a title bar containing the name of the command that created the window. This name is used to refer to specific display windows when using the **dynamic**, **size**, and **close** commands discussed below. When the output of a command requires more lines than are currently available, Fx will display as many lines a possible and then wait for you to press a key. At this point, you can press the Return or Enter key to display additional lines of output, or press any other key to terminate output display.

Setting Breakpoints

We will set a breakpoint in the tutorial program on line 40. To do this, type the following in the **Command** window:

```
      break "wrdcnt.c":40
```

and press Return. The location of the breakpoint will be marked with the letter B in the **Source** window.



```
File:wrdcnt.c   Unit:main(At Entry)  Line:35
  34  main(int argc,char *argv[])
  35  {
  36      int i;
  37      Finfo this_file;
  38      Finfo totals;
  39
  40 B    ParseOptions(&argc,argv);
  41
  42      if (argc == 1)  /* No file specified, use standard in */
  43      {
  44          this_file.file = stdin;
  45          this_file.name = 0;
  46          GetCounts(&this_file);
  47          WriteCounts(&this_file);
  48      }
  49      else
  50      {
  51          totals.word_count = 0;
  52          totals.line_count = 0;
  53          totals.char_count = 0;
  54          totals.name = "totals";
  55
```

Figure A-2
Breakpoint at line 40

The method for setting a breakpoint in a procedure is similar. For instance, to set a breakpoint on the first line of the procedure named `GetCounts`, you would type:

```
Fx> break GetCounts
```

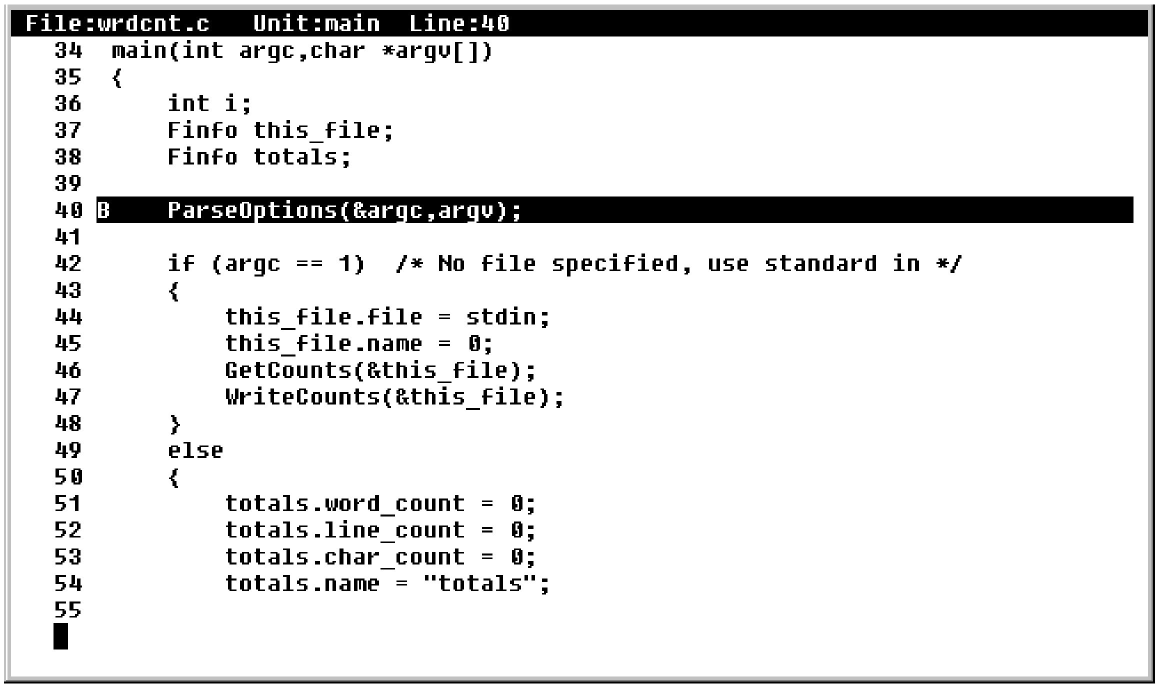
in the **Command** window and press Return. This breakpoint will also be marked with the letter B in the **Source** window.

Executing Programs

To run the `wrdcnt` program, a file needs to be specified so the program can count the words, lines and characters. This file is specified as an argument to the **run** command. For this tutorial, the number of words, lines and characters in the `wrdcnt` source file will be counted. To do this, enter the following command in the **Command** window.

```
Fx> run wrdcnt.c
```

Since we set a breakpoint, the program will stop and highlight the line where the breakpoint is set; in this case, at line 40.



```
File:wrdcnt.c   Unit:main  Line:40
 main(int argc,char *argv[])
 {
     int i;
     Finfo this_file;
     Finfo totals;

B    ParseOptions(&argc,argv);

     if (argc == 1)  /* No file specified, use standard in */
     {
         this_file.file = stdin;
         this_file.name = 0;
         GetCounts(&this_file);
         WriteCounts(&this_file);
     }
     else
     {
         totals.word_count = 0;
         totals.line_count = 0;
         totals.char_count = 0;
         totals.name = "totals";

```

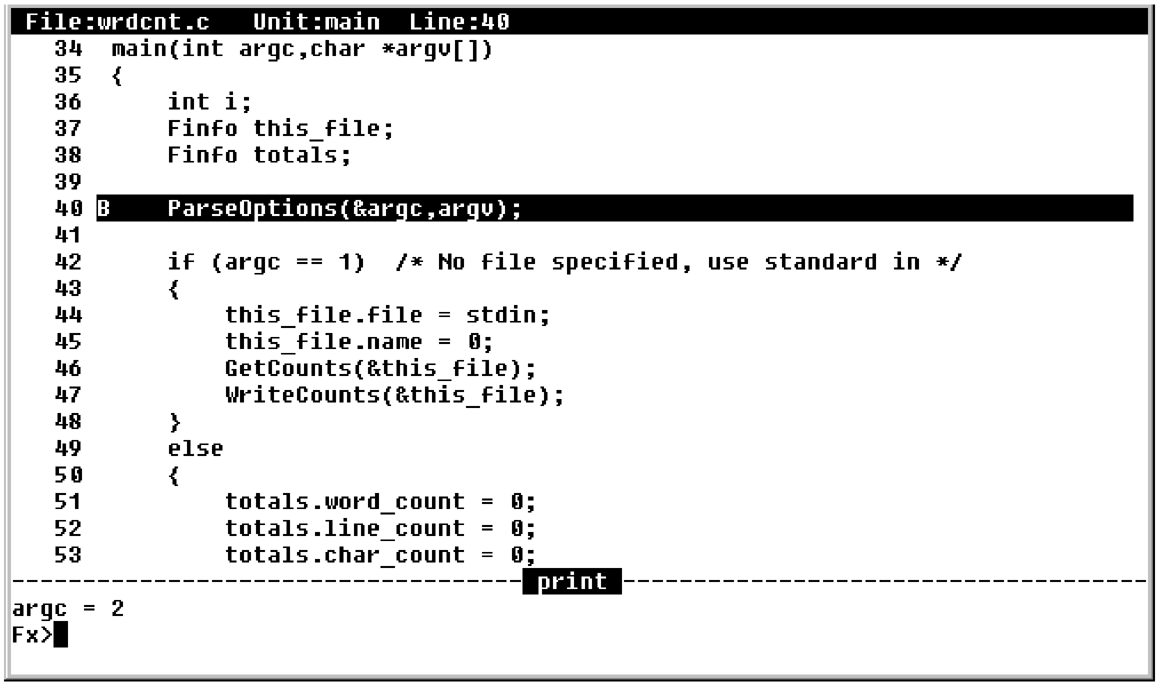
Figure A-3
Source window after issuing the run command

Printing Variables

To print the variable named `argc`, type:

```
Fx> print argc
```

and press Return. A display window will appear with the variable and its value.



```
 File:wrdcnt.c   Unit:main  Line:40
   34  main(int argc,char *argv[])
   35  {
   36      int i;
   37      Finfo this_file;
   38      Finfo totals;
   39
   40 B    ParseOptions(&argc,argv);
   41
   42      if (argc == 1)  /* No file specified, use standard in */
   43      {
   44          this_file.file = stdin;
   45          this_file.name = 0;
   46          GetCounts(&this_file);
   47          WriteCounts(&this_file);
   48      }
   49      else
   50      {
   51          totals.word_count = 0;
   52          totals.line_count = 0;
   53          totals.char_count = 0;
------------------------------------ print -------------------------------------
argc = 2
Fx>
```

Figure A-4
Print Display window

At this point, we can continue to execute the program to the next breakpoint by entering:

```
Fx> continue
```

in the **Command** window and pressing Return. The **Display** window will disappear, and the tutorial program will execute to the next breakpoint located at the start of the `GetCounts` procedure.

Executing Single Statements

To execute the program one statement at a time, use the **step** command.

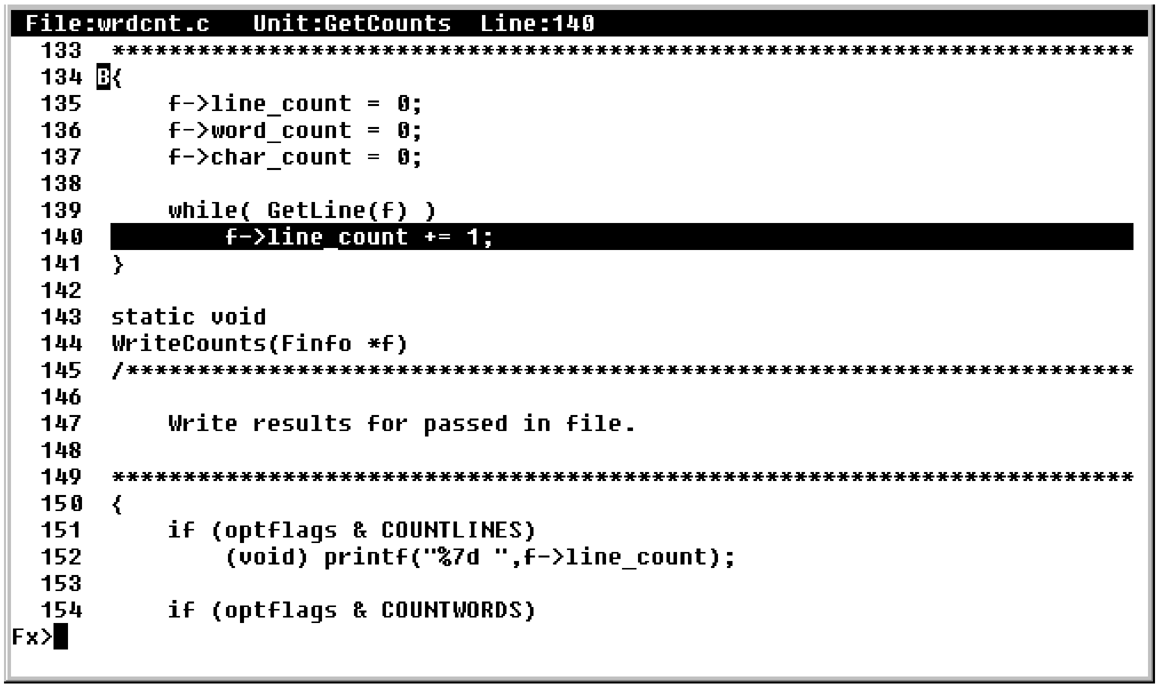
```
Fx> step
```

Since Fx will remember the previous command, type Return to execute the **step** command three more times.

The **Step** command, with a capital "S", is used to step over subroutines. To avoid following the next subroutine and stop on the line in the current procedure, type:

```
Fx> Step
```

and press Return.



```
File:wrdcnt.c   Unit:GetCounts  Line:140
************************************************************************
B{
    f->line_count = 0;
    f->word_count = 0;
    f->char_count = 0;

    while( GetLine(f) )
        f->line_count += 1;
}

static void
WriteCounts(Finfo *f)
/***********************************************************************

    Write results for passed in file.

************************************************************************
{
    if (optflags & COUNTLINES)
        (void) printf("%7d ",f->line_count);

    if (optflags & COUNTWORDS)
Fx>
```

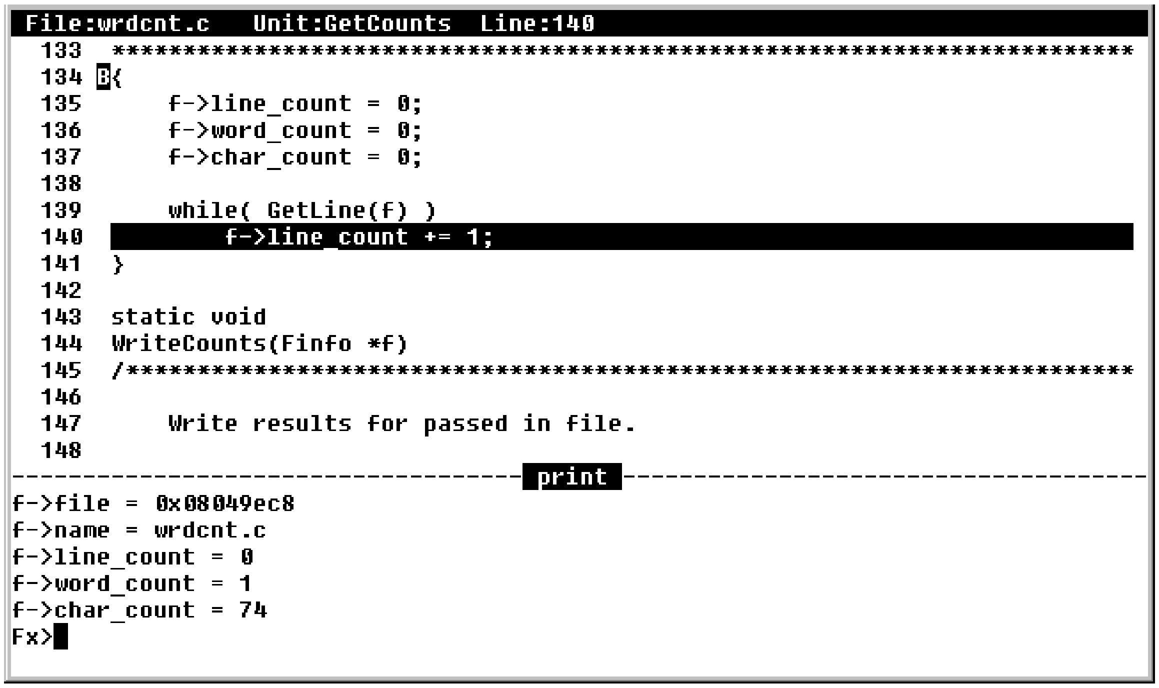
Figure A-5
After the Step command

Dereferencing C Variables

To dereference a C variable, use the **print** command with the * symbol before the variable. For example, to dereference the variable, first type at the prompt:

```
Fx> print *f
```

and hit Return.



```
 File:wrdcnt.c   Unit:GetCounts  Line:140
 ************************************************************************
B{
     f->line_count = 0;
     f->word_count = 0;
     f->char_count = 0;

     while( GetLine(f) )
         f->line_count += 1;
 }

 static void
 WriteCounts(Finfo *f)
 /************************************************************************

     Write results for passed in file.

-----------------------------------print------------------------------------
f->file = 0x08049ec8
f->name = wrdcnt.c
f->line_count = 0
f->word_count = 1
f->char_count = 74
Fx>
```

Figure A-6
Dereferencing a variable

The display window will appear, showing the number of words, lines and characters counted up to this point.

Watching Variables

To determine if a variable is being overwritten by a function, you can use the **watch** command to watch the value of the variable as the program executes.

To watch the `f>line_count` and the `f>word_count` variables:

```
Fx> watch f->line_count
Fx> watch f->word_count
```

The display window will appear which contains the values of each of the variables. Each variable is listed with an id number next to it. To watch the values as the program executes, enter:

```
Fx> Step
```

To remove the first variable that you are watching from the display window, enter the **unwatch** command with the id number of the variable.

```
unwatch 1
```

and press Return. To stop watching all variables and close the window, enter

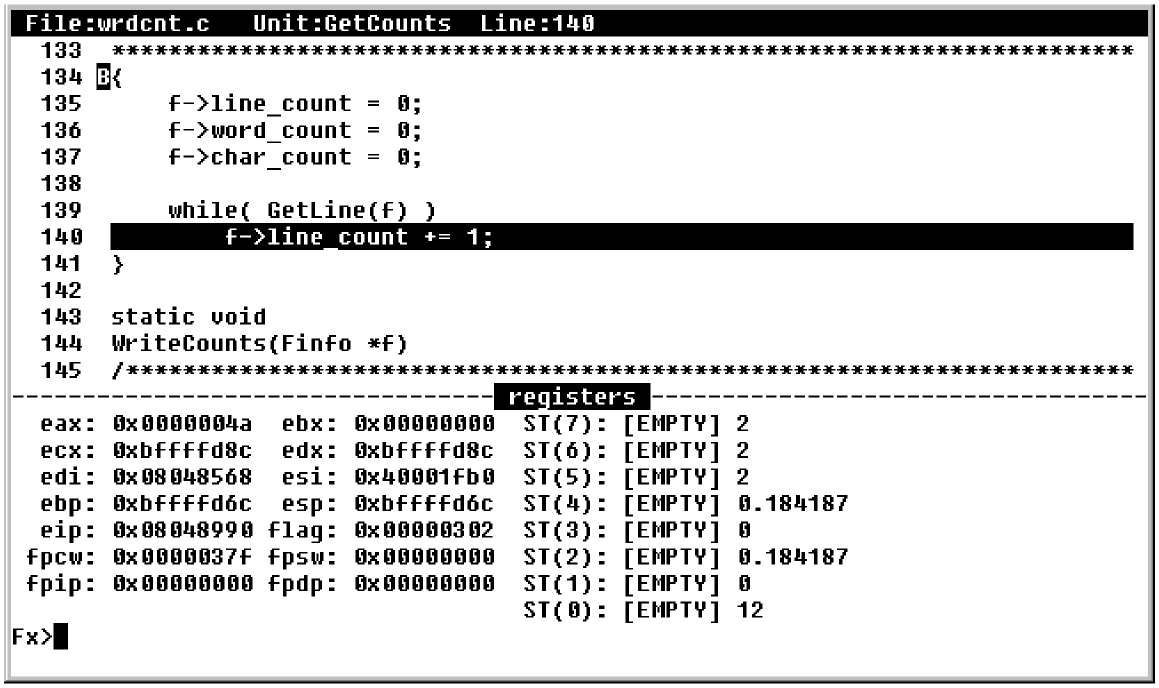
```
Fx> close watch
```

Keeping Display Windows Visible

With the exception of the windows created by **list file** and **watch** commands, display windows are automatically closed when you execute another command. Occasionally, you may want a window to remain on the screen for continuous reference. You can use the **keep** command to cause a display window to remain visible during execution of subsequent commands.

You can use the following commands to display the contents of the machine registers and have the displayed information remain on the screen during execution of further commands:

```
Fx> registers
Fx> keep
```



```
File:wrdcnt.c   Unit:GetCounts  Line:140
  ****************************************************************************
 B{
      f->line_count = 0;
      f->word_count = 0;
      f->char_count = 0;

      while( GetLine(f) )
          f->line_count += 1;
  }

  static void
  WriteCounts(Finfo *f)
  /***************************************************************************
------------------------------------registers-----------------------------------
  eax: 0x0000004a  ebx: 0x00000000  ST(7): [EMPTY] 2
  ecx: 0xbffffd8c  edx: 0xbffffd8c  ST(6): [EMPTY] 2
  edi: 0x08048568  esi: 0x40001fb0  ST(5): [EMPTY] 2
  ebp: 0xbffffd6c  esp: 0xbffffd6c  ST(4): [EMPTY] 0.184187
  eip: 0x08048990 flag: 0x00000302  ST(3): [EMPTY] 0
 fpcw: 0x0000037f fpsw: 0x00000000  ST(2): [EMPTY] 0.184187
 fpip: 0x00000000 fpdp: 0x00000000  ST(1): [EMPTY] 0
                                    ST(0): [EMPTY] 12
Fx>
```

Figure A-7
Registers display window

Making Display Windows Dynamic

You can use the **dynamic** command to have the command that created a window automatically executed after each subsequent command. If you want to watch the contents of the machine registers change as you executed your program, you can enter the following commands:

```
Fx> registers
Fx> dynamic registers
```

Resizing Display Windows

If you are using the **registers** command to examine the contents of a file, you can increase the size of the window displaying the file by three lines using the **size** command:

```
Fx> size registers,3
```

The following command will restore the window to its original size:

```
Fx> size registers,-3
```

Closing Display Windows

The **close** command removes a display window from the screen. To remove the display window created by a **print** command, you can enter:

```
Fx> close print
```

Changing the Value of Variables

The **change** command is used to change the value of variables. For example, to change the value of the variable `f->word_count`, type:

```
Fx> change f->word_count = 100
```

To see the new value, use the **print** command.

```
Fx> print f->line_count
```

Exiting from the Character Interface

To exit the debugger at any time and return to the shell prompt, enter:

```
Fx> quit
```

Using the Keyboard

In addition to entering commands in the **Command** window, you can also use certain keystrokes to perform common operations. For example, the **next** command will scroll the source window forward one page. You can also perform this action by holding down the control key and pressing the F key. If your keyboard has a page down key, you can use it as well. The table below summarizes the keystrokes and special keys you can use with the character interface.

Keystroke	Special Key	Action
CTRL-F	Page Down	Scrolls source window forward one page
CTRL-B	Page Up	Scrolls source window backward one page
CTRL-J	Down Arrow	Scrolls source window forward one line
CTRL-K	Up Arrow	Scrolls source window backward one line
CTRL-R		Recalls last command line
CTRL-U		Erases current command line
CTRL-H	Backspace	Erases last character entered on command line

Using Function Keys

If your terminal has function keys, you can assign one or more Fx commands to each key. When you want to assign a command to a function key, you use the **change** command to set the value of the appropriate function key control variable. There are 64 function key control variables named **$fk1** though **$fk64**. Other control variables are listed in Appendix B.

Use the following command to have the **continue** command executed each time the function key F1 is pressed:

```
Fx> change $fk1="continue"
```

APPENDIX B

Fx Control Variables

Fx defines a number of control variables, internal variables that modify the operation of commands. The table below lists the names of the Fx control variables, their purpose, and default values. In addition to the control variables listed below, Fx also allows up to 64 function key variables to be defined for use with the Fx character interface. The function key variables are named **$fk0** through **$fk63** and can be assigned any character string value. Unlike Fx control variables, function key variables do not exist until they are assigned a value with the **change** command.

Fx Control Variables

Name	**Controls**	**Legal Values**
$acount	maximum number of array elements displayed when unsubscripted arrays displayed with the **print** command	any positive integer, default is 100
$args	arguments passed to a program by the **run** command no arguments are specified	any character string, default is the last arguments specified with a **run** command
$case	case folding for symbol table searches	"lower", "upper", or "both", default is "both"
$cmpfmt	display format for COMPLEX values	any FORTRAN format suitable for displaying COMPLEX values, default is "('(',1PG15.E2,',',1PG15.6E2,')')"
$cwd	controls program's current working directory	a character string which specifies the directory
$dcmpfmt	display format for DOUBLE COMPLEX values	any FORTRAN format suitable for displaying COMPLEX values, default is "('(',1PG24.E3,',',1PG24.6E3,')')"
$deflang	default expression language	"C" or "FORTRAN", default is "C"

$efmt	display format for double precision values	any FORTRAN format suitable for displaying double precision values, default is "(1PG24.15E3)"

Fx Control Variables		
Name	**Controls**	**Legal Values**
$elist	whether **list entries** displays all entry points or only those with full symbol information, when non-zero all entry points are displayed	any integer, default is 0
$explang	current expression language, when set to "automatic", the expression language is determined by the current source file name	"automatic", "C", or "FORTRAN", default is "automatic"
$ffmt	display format for single precision values	any FORTRAN format suitable for displaying single precision values, default is "(1PG15.6E2)"
$glist	whether **list globals** displays all global symbols or only those with defined types, when non-zero all globals are displayed	any integer, default is 0
$leading	leading characters stripped from names in a program's symbol table	any character string, default is "_"
$lsort	whether output from **list entries**, **list globals**, **list locals**, and **list statics** is sorted, when non-zero output is sorted	any integer, default is 1
$mgrain	the number of times the command in $mstep is executed before checking monitors during execution of a **continue** or **go** command	Any positive integer, default is 1
$mstep	Fx command executed when a **continue** or **go** command is issued with active monitors	"Step","step","instruction", or "Instruction", default is "step"

Fx Control Variables		
Name	**Purpose**	**Legal Values**
$numbers	display of source line numbers when non-zero line number are displayed	any integer, default is 1
$prompt	Fx command prompt	any character string, default is "Fx>"
$read	execution of commands read from a file, when zero display commands are not executed	any integer, default is 1
$slen	maximum number of characters displayed with **s** display format	any positive integer, default is 80
$slist	whether the **list statics** command displays all static symbols or only those with defined types, when non-zero all static symbols are displayed	any integer, default is 0
$smooth	smooth scrolling of the source window, when non-zero smooth scrolling is enabled	any integer, default is 0
$tabsize	the number of spaces to expand tabs	any positive integer, default is 8
$trace	maximum number of frames to create when performing a strack trace	any positive integer, default is 50
$union	display of union members when unqualified union names are specified with the **print** command, when non-zero all members of a union are displayed	any integer, default is 1
$walkcmds	Fx commands executed by **walk** command	any list of Fx commands, default is "step"

APPENDIX C

Debugging Optimized Code

Although it is generally a sound practice to completely debug a program before turning on any compiler optimizations, there may be occasions when it is necessary to debug the optimized version of a program. While it is not impossible to debug optimized code at the source level, you should be aware that code optimization makes debugging a much harder task. For this reason, many compilers will not permit you to request optimizations when you have specified the **-g** option. This appendix describes some of the problems you will encounter when trying to debug optimized code at the source level.

Why Optimized Code is a Problem

In order for source level debugging to work, a compiler must output additional information describing the relationship between the source code it compiles and the assembly language code it produces. While the exact format of this information will vary between different compilers and operating systems, it generally assumes that there exists a one-to-one correspondence between the elements of the source program and their representation at the assembly language and machine levels.

For example, in order for a debugger to display the contents of a program variable, the compiler must output information describing the memory location where that variable will exist when the program is executed. In order to set a breakpoint, the debugger must know which assembly language instructions correspond to a particular source statement. Many of the commonly performed optimizations will dramatically alter, or even destroy, this relationship between a program and its source code.

Code Elimination Optimizations

One of the ways in which a compiler will optimize a program is remove code that has no affect on the execution of the program. For example, if a compiler can determine that the value assigned to a variable is never used during the course of a subroutine, it may not generate any code to perform the assignment. Consider the following FORTRAN subroutine:

```
SUBROUTINE SIMPLE(A,SUM)
INTEGER A(10),SUM
INTEGER I,J,K
J = 10
K = 20
SUM = 0
DO(I=1,10)
```

```
                  SUM=SUM+A(I)
            ENDDO
            RETURN
            END
```

Since the values assigned to `J` and `K` are never used anywhere in the subroutine, many compilers will not generate any code to store their values. If you use the **print** command to display either of these variables, you will see whatever happened to be present in their memory locations.

Memory and Register Optimizations

Since accessing memory is usually the slowest operation for a computer to perform, optimizing compilers will attempt to avoid accessing memory whenever possible. This is done by storing the values of variables to memory only when it is necessary. Consider the following piece of FORTRAN code:

```
DO (I=1,10)
      K = 10
      J = A(I)+K
      L = J
ENDDO
```

An optimizing compiler will probably do the following things with this loop: move the assignment of `K` above the start of the loop and move the assignment to `L` below the bottom of the loop. This transforms the source code into the following:

```
K = 10
DO(I=1,10)
      J=A(I)+K
ENDDO
L = J
```

When these optimizations are performed, Fx will not be able to display a meaningful value for L until the loop had completely executed. Another optimization that might be performed is to use keep the values of `I` and `J` in registers for the duration of the loop. Unfortunately, Fx has no way of knowing that this is occurring, and will still use the memory locations defined for these variables whenever you referenced them. For example, if you attempt to print the value of `A(I)` on the fifth execution of the loop, you will not see the fifth element of the array `A` unless `I` happened to have the value of 5 before the loop started. Also, you will not be able to use the **change** command to affect the number of times the loop executed.

Peephole Optimization

After a compiler has performed the optimizations that deal with source code, it may invoke a special optimizing pass, called a peephole optimizer, on the resulting instructions. The job of the peephole optimizer is to rearrange the generated instructions to take advantage of any hardware specific features such a branch delay slots and multiple stage pipelines. Performing this type of optimization will often cause instructions which were part of one source statement to be moved into a completely different statement, and if a program turns out to be particularly suited to this type of optimization, there will be little correspondence between the final executable code and the original source code.