ProDev™ WorkShop: Debugger User's Guide

007–2579–007

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
004	June 1998 Revised to reflect changes for the ProDev WorkShop 2.7 release.
005	April 1999 Revised to reflect changes for the MIPSpro WorkShop 2.8 release.
005	August 1999 Document released under new online and print format.
006	June 2001 Supports the ProDev WorkShop 2.9 release.
007	November 2001 Supports the ProDev WorkShop 2.9.1 release.

Contents

Figures

Tables

Examples

Procedures

About This Guide

This publication documents the ProDev WorkShop Debugger, released with the 3.0 version of ProDev WorkShop tools running on IRIX systems.

The WorkShop Debugger is a source-level debugging tool that allows you to see program data, monitor program execution, and fix code for Ada, C, C++, Fortran 77, and Fortran 90 programs.

This manual contains the following chapters:

- Chapter 1, "WorkShop Debugger Overview", page 1, gives you an introductory functional overview of the ProDev WorkShop Debugger.
- Chapter 2, "Basic Debugger Usage", page 9, outlines principles and procedures of the debugging process and how to approach them using the WorkShop Debugger.
- Chapter 3, "Selecting Source Files", page 35, describes how to manage source files.
- Chapter 4, "Tutorial: The `jello` Program", page 41, presents a short Debugger tutorial based on demonstration programs provided with your WorkShop tools.
- Chapter 5, "Setting Traps", page 65, describes how to set various types of traps.
- Chapter 6, "Controlling Program Execution", page 81, describes methods for controlling process execution.
- Chapter 7, "Viewing Program Data", page 85, explains how to examine Debugger data.
- Chapter 8, "Debugging with Fix+Continue", page 99, presents a short tutorial using Fix and Continue.
- Chapter 9, "Detecting Heap Corruption", page 115, describes heap corruption problems and how to detect them.
- Chapter 10, "Multiple Process Debugging", page 125, describes debugging multiprocess programs.
- Chapter 11, "X/Motif Analyzer", page 149, presents a short tutorial using the X/Motif Analyzer.
- Chapter 12, "Customizing the Debugger", page 165, gives you tips on how you can customize the Debugger to the requirements of your working environment.

- Appendix A, "Debugger Reference", page 171, describes all of the Debugger windows, menus, and other features in detail.
- Appendix B, "Using the Build Manager", page 329, describes the use of the Build Manager.

Related Publications

The following documents contain additional information that may be helpful:

- *C Language Reference Manual*
- *C++ Programmer's Guide*
- *MIPSpro C and C++ Pragmas*
- *ProDev Workshop: Performance Analyzer User's Guide*
- *ProDev WorkShop: Overview*
- *ProDev WorkShop: Static Analyzer User's Guide*
- *MIPSpro Fortran 77 Language Reference Manual*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*
- *MIPSpro Fortran Language Reference Manual, Volume 3*
- *MIPSpro Fortran 90 Commands and Directives Reference Manual*
- *dbx User's Guide*

Obtaining Publications

Silicon Graphics maintains publications information at the following web site:

`http://techpubs.sgi.com/library`

This library contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

To order a printed Silicon Graphics document, call 1–800–627–9307.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.

[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
GUI	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

We value your comments and will respond to them promptly.

Chapter 1

WorkShop Debugger Overview

The SGI ProDev WorkShop Debugger is a UNIX source-level debugging tool for SGI MIPS systems. It displays program data and execution status in real time. This tool can be used to debug Ada, C, C++, FORTRAN 77, and Fortran 90 programs.

For an introductory tutorial to the principles of debugging, particularly with the WorkShop Debugger, see Chapter 2, "Basic Debugger Usage", page 9.

This chapter presents an overview of the WorkShop Debugger and is divided into the following sections:

Main Debugger Features

The following sections outline the primary features and functions of the WorkShop Debugger and include references to comprehensive information found throughout this manual:

The Debugger Main View Window

When you start the Debugger with an executable file, the Main View window displays, loaded with source code, ready to execute your program with your specified arguments. Most of your debugging work takes place in the Main View window, which includes the following:

- A **menu bar** for performing debugger functions.
- A **control panel** for specifying and controlling program execution.
- A **source code display area** which displays the code for the program you are debugging.
- A **source filename** field which tells gives you the path to the file displayed in the source code display area.
- A **status area** for viewing the current status of the program.
- The Debugger **command line** in which to enter debugging commands (see "Debugger Command Line", page 313, for command syntax).

The major areas of the Main View window are shown in Figure 1-1, page 3. For a comprehensive description of the Main View window, see "Main View Window", page 171.

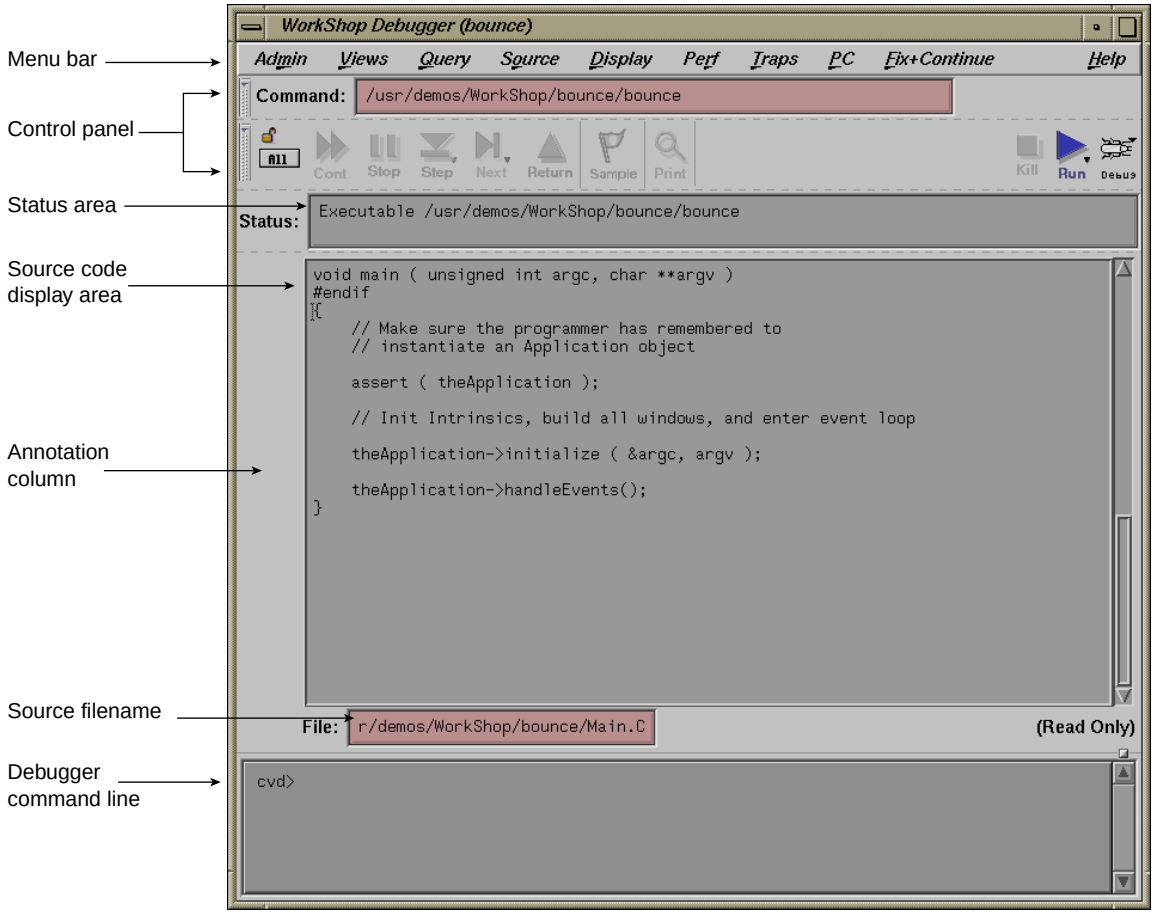

Figure 1-1 The WorkShop Debugger Main View Window

About Traps

Part of the debugging process requires that you inspect data at various points during program execution. A trap is a mechanism for gathering this data. Traps are also referred to as breakpoints, watchpoints, samples, signals, and system calls. There are two categories of traps:

- A *stop trap* halts a process so that you can manually examine data.
- A *sample trap* collects specific performance data without stopping.

The Debugger lets you set traps at the following points:

- At a line in a file (a *breakpoint*). These are the most commonly used stop traps. You can set them by clicking in the annotation column to the left of an executable statement in the source code display panel. (See the annotation column in Figure 1-1.)
- At an instruction address.
- On entry to or exit from a function.
- When a signal is received (a *signal trap*).
- When a system call is made, at either the entry or exit point (a *system call*).
- When a given variable or address is written to, read from, or executed (a *watchpoint*).
- At set time intervals (a *pollpoint*).

For more information on traps, refer to Chapter 5, "Setting Traps", page 65.

Viewing Program Data

When you stop a process, the **Views** menu at the Main View window menu bar provides several options for viewing your data. These **Views** menu options allow you to inspect the following types of data:

- **Call Stack**: this option allows you to inspect the call stack at the breakpoints.
- **Expression View**: this option allows you to inspect the value of specified expressions.
- **Variable Browser**: this option allows you to inspect the values, types, or addresses of variables.
- **Structure Browser**: this option allows you to inspect data structures.
- **Multiprocess View**: this option allows you to inspect the values of multiple and / or pthreaded processes.
- **Array Browser**: this option allows you to inspect the values of an array variable.
- **Memory View**: this option allows you to inspect the values in specified memory locations.

- **Register View**: this option allows you to inspect registers.
- **Disassembly View**: this option allows you to inspect the disassembled code.

Integrating the Debugger with Other WorkShop Tools

ProDev WorkShop tools are designed so that you can move easily between them in a work session.

Procedure 1-1 Accessing the Performance Analyzer from the Main View Window

You can run the Performance Analyzer from the Main View window as follows:

1. Select **Perf > Select Task > Task in List** from the menu bar.
2. Click on the Run button in the Control Panel. The executable is run.
3. Select **Perf > Examine Results** from the menu bar.

The Performance Analyzer window will display your results. For more information about the Performance Analyzer, see the *ProDev Workshop: Performance Analyzer User's Guide*.

Procedure 1-2 Accessing the Static Analyzer from the Main View Window

The Static Analyzer displays information that makes it easier to determine where to set traps in your source code. To launch the Static Analyzer, select **Admin > Launch Tool > Static Analyzer** from the menu bar.

For more information about the Static Analyzer, see the *ProDev WorkShop: Static Analyzer User's Guide*.

Procedure 1-3 Accessing Editors from the Main View Window

After you have isolated your code problem with the WorkShop tools, you will want to correct and recompile your source. WorkShop offers several ways to do this:

- You can select the following from the **Source** Menu to make code changes in the source pane of the Main View window:
  - Select **Source > Make Editable**. Make your changes accordingly. (**Make Editable** toggles with **Make Read Only**.)
  - Select **Source > Save** to save your changes.

  – Select **Source > Recompile** to recompile your changed code.

- You can invoke an editor from the **Views** menu that is a separate window in which to edit your code: **Views > Source View** (you must select **Make Editable** from the **File** menu at this point to proceed).
- You can call up a fork editor window to launch your own editor by using **Source > Fork Editor** (first path).
- You can call up a fork editor window to launch your own editor by using **Views > Source View** (second path) . Then, from the **Source View** window select **File > Fork Editor**.

Procedure 1-4 Accessing Configuration Management Tools

If you use ClearCase (an SGI product), RCS, or SCCS for configuration management, you can integrate the tool into the WorkShop environment by entering the following command:

```
% cvconfig [clearcase | rcs | sccs]
```

This will allow you to use **Versioning** source control (from the **Source** menu) to check files in and out.

Procedure 1-5 Recompiling from the Main View Window

You can recompile your code from the **Build View** window, accessible from the menu bar by using **Source > Recompile**.

For more information about the **Build View** window, see "Build View Window", page 329.

To examine build dependencies for your code, launch the Build Analyzer from the menu bar as follows: **Admin > Launch Tool > Build Analyzer**.

For more information about the **Build Analyzer** window, see "Build Analyzer Window", page 335.

For general information on the Build Manager tools, see Appendix B, "Using the Build Manager", page 329.

Debugging with Fix+Continue

Fix+Continue gives you the ability to make changes to a program written in C or C++ without having to recompile and link the entire program before continuing to debug the code. With Fix+Continue, you can edit a function, parse the new functions, and continue execution of the program being debugged. Fix+Continue commands may be issued from the Fix+Continue window, which you launch from the **Fix+Continue** item on the Main View menu bar.

You can also issue Fix+Continue commands from the Debugger's command line.

See Chapter 8, "Debugging with Fix+Continue", page 99, for a comprehensive description of the theory and operation of Fix+Continue, including a short tutorial.

Debugging with the X/Motif Analyzer

The X/Motif Analyzer provides specific debugging support for X/Motif applications. The X/Motif analyzer is integrated with the Debugger. You issue X/Motif analyzer commands graphically from the X/Motif analyzer subwindow of the Debugger Main View. Select **Views > X/Motif Analyzer** from the Main View window to access this subwindow.

See Chapter 11, "X/Motif Analyzer", page 149 for a comprehensive description of the theory and operation of the X/Motif Analyzer, including a short tutorial.

Customizing the Debugger

WorkShop provides you with a number of ways that you can customize your Debugger as best suited to the needs of your development environment.

See Chapter 12, "Customizing the Debugger", page 165, for more tips on customizing the Debugger to your specific needs.

Basic Debugger Usage

The WorkShop Debugger can be used with the following compilers: C, C++, Ada, FORTRAN 77, and Fortran 90.

This chapter includes information regarding principles and procedures of the debugging process and how these are to be approached with the WorkShop Debugger.

Getting Started with the Debugger

Before starting a Debugger session from a remote workstation, you must first enter the following from a window:

```
% xhost + machine_name
```

machine_name is the name or IP address of the machine where the program that you would like to debug will be run.

On the machine where your program will be running, enter the following:

```
% echo $DISPLAY
```

The *machine_name* of your workstation should appear, followed by `:0.0`. If it does not, enter the following on the machine where your program will run (if you are using the `csh` or `tcsh` shells):

```
% setenv DISPLAY machine_name:0.0
```

For other shells, see their respective man pages.

Basic Tips and Features

To provide debugging information to the Debugger, compile your program with the `-g` option (this disables optimization and produces information for symbolic debugging).

To begin a Debugger session enter:

```
% cvd executable &
```

The Debugger Main View window will automatically appear along with an icon for the Execution View window.

If your program requires data to be read from a file named `input`, for example, then type the following in the command (`cvd`) pane of the Main View window:

```
cvd> run<input
```

(See Figure 1-1, page 3.)

The **Execution View** window receives all the output from running your program that would normally go directly to your screen. The Main View window controls your Debugger session, displaying your source code in its center pane. If you want to see the line numbers for your program, select **Display > Show Line Numbers** from the Main View window menu bar.

Context sensitive help is enabled by default. This will pop-up a help phrase or statement for some menu items, data entry fields, and buttons. It can be enabled and disabled by selecting **Display > Show Tooltips / Hide Tooltips** from the Main View window menu bar.

At the bottom of the Main View window you can enter `dbx`-style commands to the Debugger. See Appendix A, "Debugger Reference", page 171, for details about which commands are supported.

The Debugger allows you to run your program and stop at selected places so you can view current values of program variables to help you find bugs in your program. To stop at a selected statement in your program, you may either set a breakpoint (also called a stop trap) at the desired statement or set a breakpoint prior to the desired statement and then use either the **Step** or **Next** buttons to reach the desired stopping point. The statement where your program has stopped is indicated in green. A statement highlighted in red indicates that a breakpoint has been set on this line. When the Debugger causes your program to stop at a breakpoint that you have set, the executable statement immediately prior to the breakpoint has been executed, and the executable statement on which the breakpoint has been set has yet to be executed.

Programs featured in this chapter are located in the `/usr/demos/WorkShop/getstarted` directory.

You can find short Debugger tutorials in "Fortran 90 Code Example and Short Tutorial", page 11 and "C Example and Short Tutorial", page 13.

See also Appendix A, "Debugger Reference", page 171 for a comprehensive description of Debugger functions.

Fortran 90 Code Example and Short Tutorial

Use the `prog.f` and `dot.f` files in `/usr/demos/WorkShop/getstarted` to demonstrate the Debugger features in the following tutorial.

Example 2-1 Fortran 90 Example

- The Fortran 90 code in the file `prog.f` is as follows:

```
        program prog
        parameter ( n=3 )
        double precision A(n,n), x(n), y(,) sum xydot

!       initialize arrays
        x = 2.0d0
        do i = 1, n
           y(i) = i
           do j = 1, n
              A(i,j) = i*j - i
           enddo
        enddo

!       compute the dot product of x and y
        call dot(x,y,n,xydot)
        print *, 'dot product of x & y = ', xydot

!       compute y = Ax
        do i = 1, n
           sum = 0.0
           do j = 1, n
              sum = sum + A(i,j)*x(j)
           enddo
           y(i) = sum
        enddo
        print *, 'y = ', y
        stop
        end
```

- It includes subroutine `dot` in file `dot.f`, as follows:

```
        subroutine dot(a,b,m,answer)
        double precision a(m), b(m), answer
        integer m
```

```
answer = 0.0d0
do i = 1, m
   answer = answer + a(i)*b(i)
enddo
end
```

Perform the following steps with these files to demonstrate Debugger features:

1. Enter the following command in the `/usr/demos/WorkShop/getstarted` directory to produce the executable program:

   ```
   % f90 -g -o progf prog.f dot.f
   ```

   This produces the executable `progf`.

2. Launch the WorkShop Debugger with your newly-compiled executable as follows:

   ```
   % cvd progf &
   ```

   The **WorkShop Debugger** Main View displays the source for your `prog.f` file (see Example 2-1, page 11).

3. Select **Display > Show Line Numbers** from the Main View menu bar to turn on file line numbering.

   The line numbers will display to the left of the source code.

4. Enter a breakpoint at line **15** as follows at the **cvd>** prompt at the bottom of the Main View window. This will enable you to execute through the end of the initialization of the `y` array for the sample code:

   ```
   cvd> stop at 15
   ```

   Line **15** is highlighted in red and a stop icon appears in the left column.

5. Run the program. There are two ways that you can do this:

   a. Click on the **Run** button at the top of the Main View window.

      OR

   b. Enter the following at the **cvd>** prompt:

      ```
      cvd> run
      ```

   The program executes up to Line **15** and waits for further instruction.

6. Enter the following command at the **cvd>** prompt to print the `y` array for this example:

   ```
   cvd> print y
   ```

   The following displays in the `cvd>` command pane:

   ```
   y =
       (1) = 1.0
       (2) = 2.0
       (3) = 3.0
   cvd>
   ```

   Note: You should expand this pane (or use the slider at the right side of the pane) if you do not see the printout.

7. At this point, you can experiment with other commands described in this chapter, notably the execution control buttons described in "Options for Controlling Program Execution", page 16.

8. Select **Admin > Exit** to end this tutorial for the Fortran 90 demo program.

C Example and Short Tutorial

Use the `prog.c` and `dot.c` files in `/usr/demos/WorkShop/getstarted` to demonstrate the Debugger features in the following tutorial.

Example 2-2 C Code Example

The following is the same example as that in "Fortran 90 Code Example and Short Tutorial", page 11, but it is written in C. Use this example to see how the Debugger can be used to view C structures.

- The C code in the file `prog.c` is as follows:

```
#include <stdio.h>
#define  N  3
double dot(double v[],double w[], int m);
void main(){
  int i,j;
  double a[N][N],x[N],y[N],sum,xydot;
  struct node
```

```
{
  int value;
  struct node *next;
} *list,start;

/* Initialize arrays */
for(i=0;i<N;i++){
  x[i]=2;
  y[i]=i;
  for(j=0;j<N;j++){
    a[i][j]=i*j-i;
  }
}

/* Compute the dot product x and y */
xydot=dot(x,y,N);
printf("dot product of x & y: %f \n",xydot);

/* Compute y=ax */
for(i=0;i<N;i++){
  sum=0;
  for(j=0;j<N;j++){
    sum+=a[i][j]*x[j];
  }
  y[i]=sum;
}
printf("y = ");
for(i=0;i<N;j++){
  printf("%f ",y[i]);
}
printf("\n");

/* Built list*/
start.value=1;
list=&start
for(i=1;i<N;i++){
list->next=(struct node *) malloc(sizeof(struct node));
list=list->next;
list->value=i;
}
list->next=NULL;
```

```
  printf("list: ");
  list=&start;
  for(i=0;i<N;i++){
    printf("%d ",list->value);
    list=list->next;
  }
  printf("\n");
}
```

- It includes function `dot` in file `dot.c`, as follows:

```
double dot(double v[],double w[], int m){
  int i;
  double sum;
    for(i=0;i<m;i++){
    sum+=v[i]*w[i];
  }
  return(sum);
}
```

Perform the following steps with this file to demonstrate Debugger features:

1. Enter the following command in the `/usr/demos/WorkShop/getstarted` directory to produce the executable program:

   ```
   % cc -g -o progc prog.c dot.c
   ```

   This produces the executable `progc`.

2. Launch the WorkShop Debugger with your newly-compiled executable as follows:

   ```
   % cvd progc &
   ```

   The **WorkShop Debugger** Main View displays the source for your `prog.c` file (see Example 2-2, page 13).

3. Select **Display > Show Line Numbers** from the Main View menu bar to turn on file line numbering.

   The line numbers will display to the left of the code source window.

4. Enter a breakpoint at line **23** as follows at the **cvd>** prompt at the bottom of the Main View window. This will enable you to execute up to the end of the `y` array for the sample code:

   ```
   cvd> stop at 23
   ```

   Line **23** is highlighted in red and a stop icon appears in the left column.

5. Run the program. There are two ways that you can do this:

   a. Click on the **Run** button at the top of the Main View window.

      OR

   b. Enter the following at the **cvd>** prompt:

      ```
      cvd> run
      ```

   The program executes up to Line **23** and waits for further instruction.

6. Enter the following command at the **cvd>** prompt to print the `y` array for this example:

   ```
   cvd> print y
   ```

   The following displays in the `cvd` command pane:

   ```
   y = {
       [0] 0.00000000000000000e+00
       [1] 1.0
       [2] 2.0
   }

   cvd>
   ```

7. At this point, you can experiment with other commands described in this chapter, notably the execution control buttons described in "Options for Controlling Program Execution", page 16.

8. Select **Admin > Exit** to end this tutorial for the C demo program.

Options for Controlling Program Execution

There are a number of buttons in the Main View window that allow you to control the execution of your program. The following summarizes their functions:

- **Run** creates a new process to execute your program and starts execution. It can also be used to rerun your program.
- **Kill** kills the active process that is executing your program.
- **Stop** stops execution of your program. The first executable statement after the statement where your program has stopped is highlighted.
- **Cont** continues program execution until a breakpoint or some other event stops execution, or program execution terminates. (See also "How to Continue a Single POSIX 6.5 Pthread", page 143.)
- **Step** steps to the next executable statement and into function and subroutine calls. Thus, if you set a breakpoint at a subroutine call, click on the **Run** button so the call to the subroutine is highlighted in green, then click on the **Step** button to step into this subroutine — source code for this subroutine will automatically be displayed in the Main View window.

  By clicking the right mouse button on the **Step** button you can select the number of steps the Debugger takes. Left-click on the **Step** button to take one step.
- **Next** steps to the next executable statement and steps over function and subroutine calls. Thus, if you set a breakpoint at a subroutine call, click on the **Run** button so the call to the subroutine is highlighted in green, then click the **Next** button to step over this subroutine to the next executable statement displayed in the source pane of the Main View window.

  Right-click on the **Next** button to select the number of steps the Debugger takes. Left-click on the **Next** button to take one step.
- **Return** executes the remaining instructions in the current function or subroutine. Execution stops upon return from that function or subroutine.
- **All or This** applies control action to all processes or threads if the button is set to **All**. If set to **This**, the actions apply only to this process or thread.
- **Lock** causes the debugger to stay focused on this process or thread, no matter what the program does. If unlocked, the debugger follows the interesting process or thread (i.e., focuses on a process or thread that reaches a breakpoint/trap).

Setting Traps

A trap (also called a *breakpoint*) can be set if you click your cursor in the area to the left of the statement in the area underneath the word **Status** in the Main View

window. When you do this, the line in your program will be highlighted in red. To remove the breakpoint, click in the same place and the red highlighting will disappear. Clicking on the **Run** button will cause your program to run and stop at the first breakpoint encountered. To continue program execution, click on the **Continue** button. Breakpoints can only be set at executable statements.

Options for Viewing Variables

There are many ways to view current values of variables with the Debugger. Before you can do this, you must first run your program under the Debugger and stop execution at some point. The following sections list ways of viewing current values of variables with the Debugger. It is suggested that you try each of the following methods to determine which you would prefer to use.

Viewing Variables Using the `cvd` Command Line

At the bottom of the Main View window is the command/message pane. Here, you can give the Debugger various instructions at the `cvd` command line.

Example 2-3 Value of array *x*

If you want to know the current value of array *x*, enter either of the following two commands in the command/message pane:

```
cvd> print x
```
or

```
cvd> p x
```

The current values for *x* will be printed.

Example 2-4 Value of *x*(2)

If your program is written in Fortran and you only want the value of *x*(2), enter:

```
cvd> print x(2)
```

For a C program, enter:

```
cvd> print x[2]
```

Example 2-5 Change value of *x*(2) to 3.1

To change the value of *x*(2) to 3.1 in a Fortran program, enter:

```
cvd> assign x(2) = 3.1
```

For a C program, enter:

```
cvd> assign x[2] = 3.1
```

The value of *x*(2) will now be 3.1 when execution is resumed.

Such changes are only active during the current Debugger run of your program. In the preceding examples, if *x* is a large array, you may want to use the **Array Browser** window (see "Viewing Variables Using the **Array Browser**", page 22).

To view the components of the structure `start` in the `prog.c` example, enter:

```
cvd> print start
```

The current values of each component of `start` will be printed.

To view what the pointer `list` points to in the `prog.c` example, enter:

```
cvd> print *list
```

This pointer must be initialized before you can perform this function.

A complete list of the instructions that can be entered at the `cvd` command line can be found in "Debugger Command Line", page 313.

Viewing Variables Using `Click To Evaluate`

Perform the following to view variables with `Click To Evaluate`:

1. Right-click in the window that contains your source code to bring up the following pop-up menu:

Figure 2-1 Evaluation Pop-Up Menu

2. Select **Click To Evaluate** from this menu. You can now click on any variable and its value will appear. For example:
   - If you click on the `x` in `x(i)`, the address of `x` will appear.
   - If you click-drag to highlight `x(i)`, the current value of `x(i)` displays.
   - If you highlight an expression, the current value of the expression displays.

Viewing Variables Using the Variable Browser

Perform the following to view variables with the `Variable Browser`:

1. Select **Views > Variable Browser** from the Main View window to call up the Variable Browser.

   The **Variable Browser** automatically displays values of all variables valid within the routine in which you have stopped as well as the address location for each array.

   Values of variables can be changed by typing in their new value in the **Result** column and then hitting the `ENTER` key (or `RETURN` key, for some keyboards).

Viewing Variables Using the Expression View Window

The **Expression View** window allows you to enter the variables and/or expressions for which you would like to know values.

1. Select **Views > Expression View.** from the Main View window menu bar to display the **Expression View** window.

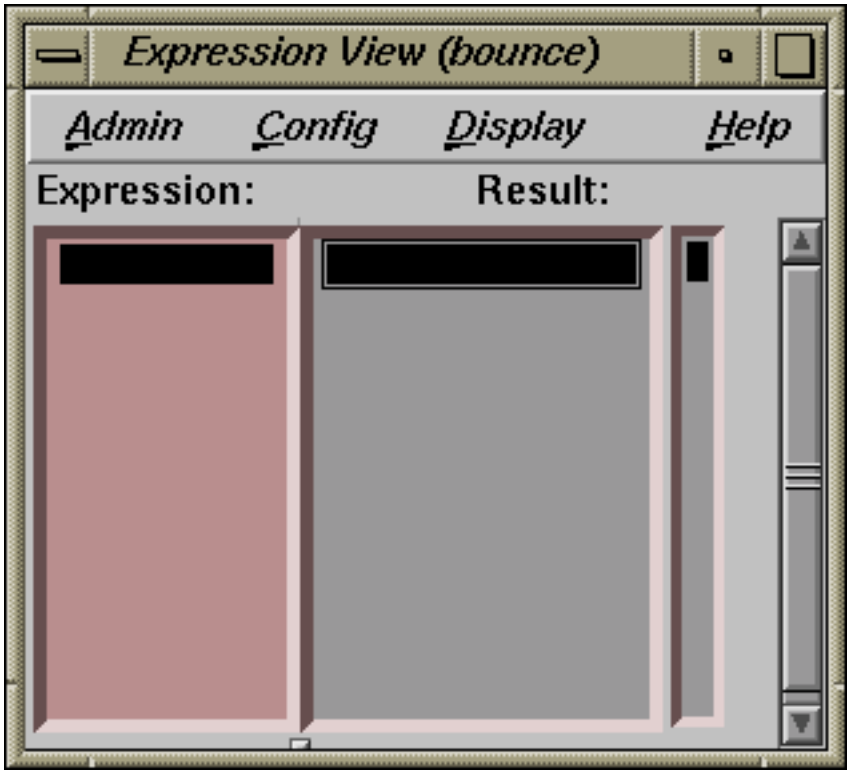

Figure 2-2 Expression View Window

2. To view, for example, the **value** component of the structure **start** in `prog.c`, enter the following in the **Expression** column:

   ```
   start.value
   ```

   As you step through your program, for example with **Step Over**, the values of all the entries in this window will be updated to their current values.

   Values of variables can be changed by typing in their new value in the **Result** column and then pressing ENTER.

To enter an expression from your source code into the **Expression View** window:

1. Left-click and drag on the expression in your source code. The expression is highlighted.

2. Left-click in a field in the **Expression** column. The cursor appear in the field.
3. Middle-click your mouse. The desired value appears in the field.

Viewing Variables Using the Array Browser

To view array values, select **Views > Array Browser** from the Main View window menu bar to call up the **Array Browser**.

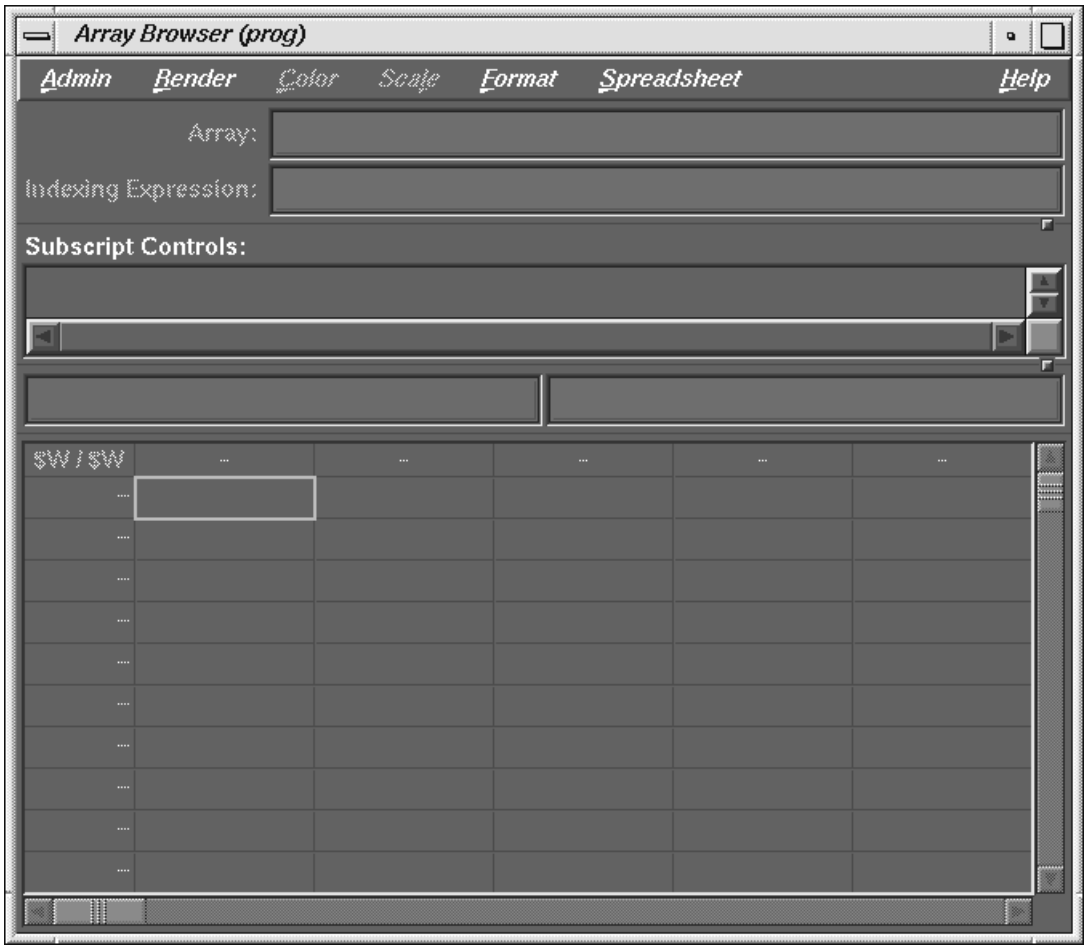

Figure 2-3 Array Browser Window

To view the values of an array in the **Array Browser**:

1. Enter the name of this array in the **Array** field
2. Press **ENTER**

Here, the current values of, at most, two dimensions of the array will display in the lower pane of the **Array Browser** window. The values of the array will be updated to their current values as you step through your program.

If the array is large or has more than two dimensions, the **Subscript Controls** panel in the middle of the **Array Browser** window allows you to specify portions of the array for viewing. You may also use the slide bars at the bottom and right of the window to view hidden portions of an array.

Procedure 2-1 Changing values of array elements

Perform the following to change values of array elements:

1. Click on the box with the array value in the lower portion of the **Array Browser** window. After the element is selected, the array index and value appear in the two fields below the **Subscript Controls** panel in the center of the **Array Browser** window.

   For example, if you click on the value for array element A(2,3), then **A(2,3)** appears in the box above the display of the array values, and its value appears in the box to the right of A(2,3). Simply click in this box, and enter a new value for A(2,3). (Press `ENTER` to change the value of A(2,3) to the new value.)
2. Enter your change into the `Value` field described in the note above.
3. If you would like to view a second array at the same time, select **Admin > Clone** in the **Array Browser** window that you have already opened. This brings up a second **Array Browser** window.
4. Select **Admin > Active** from this new window.

You can now enter the name of the second array you would like to view.

Procedure 2-2 Viewing values of a C structure

Perform the following to view values of a C structure:

1. Select **Views > Structure Browser** from the Main View window menu bar to call up the **Structure Browser** window.

2. Enter the name of the structure in the **Expression** field. This brings up a window listing the name of the structure, the names of its components in the left column, and their values in the right column.
3. If one of these components is a pointer, you can see what is being pointed to by double-clicking on its value to the right of the pointer name. This will bring up a new window showing what is being pointed at and an arrow will appear showing this relationship. This can aid in debugging linked lists, for example.

Searching

Often it is useful to search for the occurrences of a variable or some character string in your program so you can set breakpoints. Perform a search as follows:

1. Call up the search utility from the Main View window menu bar using **Source > Search**.

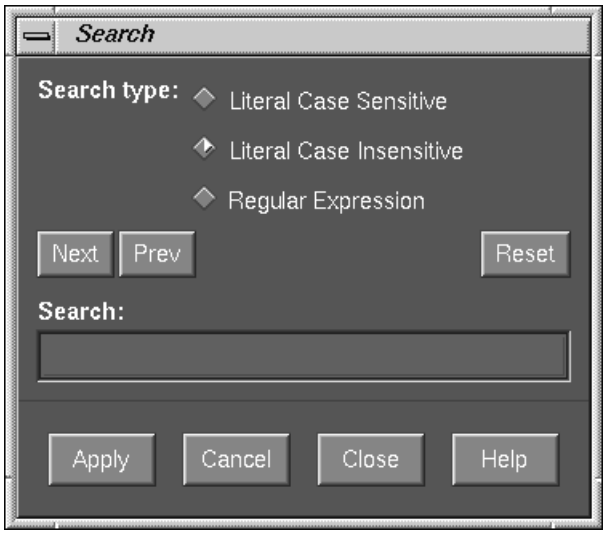

Figure 2-4 Search Window

2. Enter the character string for which you would like to search in this window, then click on the `Apply` button. All occurrences of this string are highlighted.

3. Click on **Cancel** to remove highlighting.

Using the Call Stack

As the Debugger executes, it may stop during a program function or subroutine, or in a system routine. When this happens, you can locate where the Debugger is in your program by examining the call stack.

There are two ways to examine the call stack:

- You can enter the following in the command/message pane:

  ```
  cvd> where
  ```

  This lists the functions and subroutines that were called to bring the Debugger to this place. This call list includes not only your program functions/subroutines but may also include system routines that have been used. You can now move up or down the call tree by issuing, for example:

  ```
  cvd> up [n]
  ```

  In this case, the source code for a function or subroutine that is 'up' *n* items in the call stack will appear in the Main View window. If you omit **[*n*]** you move up one item in the call stack.

- You can select **Views > Call Stack** from the Main View window menu bar. This brings up the **Call Stack** window.

Figure 2-5 Call Stack Window

If you double-click on an item here, the source code for the function or subroutine, if available, will display in the Main View window.

Stopping at Functions or Subroutines

In the debugging process, it is sometimes useful to stop at each occurrence of a function or subroutine. The Debugger permits you to do this in either of two ways:

- Using the `cvd` command/message pane.

  1. Enter the following in the command/message pane of the Main Window:

     `cvd> stop in` *name*

     For *name*, specify the name of the function or subroutine in your program where you would like the Debugger to stop.

  2. Click on the `Run` button and the Debugger will stop each time it encounters this function or subroutine.

  3. To remove this stopping condition enter the following in the command/message pane:

     `cvd> status`

For the `Stop in` command above, the trap in the list would appear as:

`[`*n*`] Stop entry` *name*`...`

Here, the value of *n* is a positive integer and *name* is the name of the function or subroutine where the stop has been set.

4. To delete this stop, enter:

   `cvd> delete` *n*

- Using the Trap Manager.

  1. Select **Views > Trap Manager** from the Main View window menu bar to use the trap manager. This calls the **Trap Manager** window.

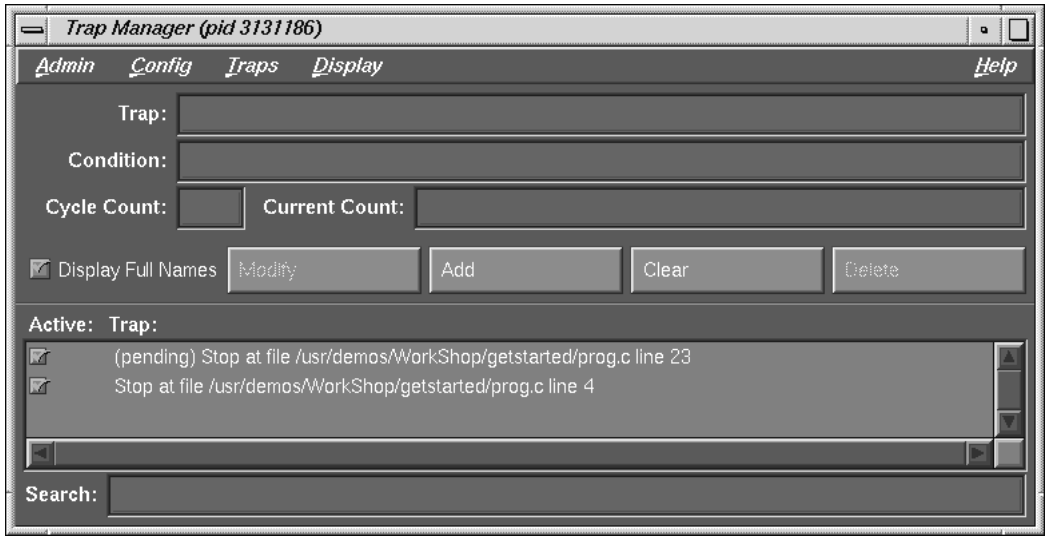

Figure 2-6 Trap Manager Window

  2. In the text field to the right of the word **Trap** enter:

     `stop in` *name*

3. Click on the **Add** button or press **Enter**. This adds your breakpoint at your desired function or subroutine.

To remove the stopping condition so the Debugger will not stop at each occurrence of *name*, click on the **Delete** button in the **Trap Manager** window.

If you have multiple traps displayed, click on the trap that you wish to delete before you click on the **Delete** button.

Suggestions for Debugging for Serial Execution of Scientific Programs

This section offers tips and suggestions for debugging programs written for scientific applications; but many of the suggestions will apply to debugging other types of applications as well.

Note: This section deals only with debugging programs that are running *serially* and not in parallel.

Programs can sometimes appear to have no bugs with some sets of data because all paths through the program may not be executed. To debug your program, therefore, it is important to test it with a variety of different data sets so that, one would hope, all paths in your program can be tested for errors.

Now, assume that your program compiles and produces an executable, but the program execution either does not complete, or it completes but produces wrong answers. In this case, go through the following steps to find many of the commonly occurring bugs:

All compiler options mentioned in the following sections are valid for FORTRAN 77, Fortran 90, C and C++ compilers unless indicated otherwise.

Step 1: Use `lint`

If your program is written in C, you should use the `lint` utility. This will help you identify problems with your code at the compile step. For example, if your C program is in a file named `prog.c`, invoke `lint` with:

```
> lint prog.c
```

The output from this command is directed to your screen.

There is a public domain version of `lint` for FORTRAN 77 called `ftnchek`. You can get `ftnchek` from the `/pub` directory at the anonymous `ftp` site `ftp.dsm.fordham.edu` at Fordham University.

Step 2: Check for Out-of-Bounds Array Accesses

A common programming error is the use of array indices outside their declared limits. For help finding these errors, compile your program as follows:

```
> -g -DEBUG:subscript_check=ON
```

Then run the generated executable under `cvd` and click on the **RUN** button. (See the `DEBUG_group`(5) man page for more information on this option.)

The following list explains compiling dependencies when working with out-of-bounds array accesses:

- If you are running a C or C++ program, your program will stop at the first occurrence of an array index going out of bounds. You can now examine the value of the index that caused the problem by using any of the methods described in "Options for Viewing Variables", page 18. If you compile with the `-g` option, the compiler generates symbolic debugging information so your program will execute under `cvd`. It also disables optimization. Sometimes, disabling optimization will cause the bug to disappear. If this happens, you should still carefully go through each of these steps as best as you can.
- If you are using the Fortran 90 compiler, after compiling with the preceding options and after running the generated executable under `cvd`, enter the following in the `cvd` pane:

  ```
  cvd> stop in __f90_bounds_check
  ```

  Note: `__f90` in this command has two lead "_" characters.

Now, click on the **RUN** button. Next select **Views > Call Stack** from the Main View window menu bar.

Next, double click on the function or subroutine immediately below **__f90_bounds_check**. This will cause the source code for this function or subroutine to display in the Main View window, and the line where `cvd` has stopped will be highlighted. You can now find the value of the index that caused the out-of-bounds problem.

- If you are using the FORTRAN 77 compiler, after compiling with the preceding options and after running the generated executable under `cvd`, enter the following in the `cvd>` pane:

  ```
  cvd> stop in s_rnge
  ```

  Click on the **RUN** button. Next, select **Views > Call Stack** from the Main View menu bar.

  Double-click on the function or subroutine immediately below `s_rnge`. This will cause the source code for this function or subroutine to display in the Main View window and the line where the Debugger stopped will be highlighted. You can now find the value of the index that caused the out-of-bounds problem.

Note: For Fortran programs, bounds checking cannot be performed in subprograms if arrays passed to a subprogram are declared with extents of `1` or `*` instead of passing in their sizes and using this information in their declarations. An example of how the declarations should be written to allow for bounds checking is: `SUBROUTINE SUB(A,LDA,N, ...)  INTEGER LDA,N REAL A(LDA,N)`

Step 3: Check for Uninitialized Variables Being Used in Calculations

To find uninitialized `REAL` variables being used in floating-point calculations, compile your program with the following:

```
-g -DEBUG:trap_uninitialized=ON
```

This forces all uninitialized stack, automatic, and dynamically allocated variables to be initialized with `0xFFFA5A5A`. When this value is used as a floating-point variable involving a floating-point calculation, it is treated as a floating-point NaN and it causes a floating-point trap. When it is used as a pointer or as an address a segmentation violation may occur. For example, if `x` and `y` are real variables and the program is compiled as described previously, `x = y` will not be detected when `y` is

uninitialized since no floating point calculations are being done. However, the following will be detected:

```
x = y + 1.0
```

After you compile your program with the preceding options, enter the following:

% **cvd** *executable*

Then click the **RUN** button. To find out where your program has stopped, select **Views > Call Stack** from the Main View window menu bar.

Here, you will see that many routines have been called. Double-click on the routine closest to the top of the displayed list that is not a system routine. (You will probably recognize the name of this source file.) This will bring up the source code for this routine and the line where the first uninitialized variable (subject to the above-mentioned conditions) was used. You can now examine the values of the indices which caused the problem using any of the methods described in "Options for Viewing Variables", page 18. You cannot use `cvd` to detect the use of uninitialized `INTEGER` variables.

Step 4: Find Divisions by Zero and Overflows

If you are using a csh or tsch shell, perform the following to find floating-point divisions by zero and overflows (for other shells, see their man pages for the correct command).

1. Enter the following:

   % **setenv TRAP_FPE ON**

2. Compile your program using the following options:

   compiler command **-g -lfpe**

3. Enter the following:

   % **cvd** *executable*

4. In the `cvd` command/message pane enter:

   cvd> **stop in _catch**

5. Click on the **RUN** button.

6. Select **Views > Call Stack** from the Main View window.
7. Double-click on the routine closest to the top of the displayed list that is not a system routine. (You will probably recognize the name of this source file.)

   The line where execution stopped is highlighted in the source code display area of the Main View window.

   You may now use any of the methods to find variable values, described in "Options for Viewing Variables", page 18, to discover why the divide-by-zero or overflow occurred.

For more information on handling floating-point exceptions, see the `handle_sigfpes`(3) and `fsigfpe`(3f) man pages.

Perform the following to find integer divisions by zero:

1. Compile your program using the following options:

   ```
   -g -DEBUG:div_check=1
   ```

2. Enter the following:

   ```
   % cvd executable
   ```

3. Click the **Run** button.

   The program automatically stops at the first line where an integer divide-by-zero occurred. You may now use any of the methods to find variable values, described in "Options for Viewing Variables", page 18, to discover why the divide-by-zero occurred.

Step 5: Perform Core File Analysis

Sometimes during program execution a core file is produced and the program does not complete execution. The file is placed in your working directory and named `core`.

Some machines are configured to not produce a core file. To find out if this is the case on the machine you are using, enter the following:

```
% limit
```

If the limit on `coredumpsize` is zero, no corefile will be produced. If the limit on `coredumpsize` is not large enough to hold the program's memory image, the core file produced will not be usable.

To change the configuration to allow core files to be produced enter the following:

```
% unlimit coredumpsize
```

After you have a core file, you can perform the following analysis:

1. You can find the place in your program where the execution stopped and the core file was produced by entering:

   ```
   % cvd executable core
   ```

   Here, *executable* is the executable that you were running.

   The Main View window comes up and the source line where execution stopped may be highlighted in green.

2. If the source line is not highlighted in green, select **Views > Call Stack** from the Main View window menu bar.

3. Double-click on the routine closest to the top of the displayed list that is not a system routine. (You will probably recognize the name of this source file.) This will bring up the source code for this routine, and the last line executed will be highlighted in green.

If the executable was formed by compiling with the `-g` option, then you can view values of program variables when program execution stopped.

To find the assembly instruction where execution stopped, select **Views > Disassembly View** from the Main View window menu bar.

Remember that this is the last statement executed before the `core` file was produced. It therefore does not necessarily mean that the bug in your program is in this line of code. For example, a program variable may have been initialized incorrectly; but the core was not produced until the variable was used later in the program.

Step 6: Troubleshoot Incorrect Answers

Assume that the preceding steps have been taken and that all detected problems have been corrected. Your program now completes execution, but obtains incorrect answers. What you do at this point will likely depend on special circumstances. The following is a list of some commonly used debugging tips that may or may not apply to your situation.

1. Try running your program on a very small size problem where you can easily obtain intermediate results. Run your program under `cvd` on this small problem and compare with the known correct results.
2. If you know that a certain answer being calculated is not correct, set breakpoints in your program so you can monitor the value of the answer at various points in your program.
3. You may want to set breakpoints on each call to a selected function or subroutine where you suspect there may be problems. (See"Options for Viewing Variables", page 18 for suggested methods.)
4. Debug `COMMON` blocks and `EQUIVALENCE` statements in Fortran. Variables used in these statements must have the same type and dimension everywhere they appear and they must occur in the same order. Normally `ftnchek`, for FORTRAN 77 programs, and `cflint`, for Fortran 90 programs, will find these errors. However, for FORTRAN 77 programs it is best to use an include statement for each `COMMON` block. For Fortran 90 programs, it is best to use a module for each `COMMON` block. It is best not to use `EQUIVALENCE` statements.
5. Save local data that is otherwise not saved. In Fortran, values of local variables are not guaranteed to be saved from one execution of the subprogram to the next unless they are either initialized in their declarations or they are declared to have the `SAVE` attribute. Some compilers and machines automatically give all local variables the `SAVE` attribute, so moving a working program from one compiler or machine to a compiler or machine that does not do this may cause this bug to manifest. The Fortran standards require that you give all uninitialized local variables the `SAVE` attribute if you would like their values saved.

Selecting Source Files

This chapter shows you how to select source files for the source pane of the Main View window of the Debugger (see Figure 1-1, page 3). It covers the following topics:

How to Load Source Files

The following sections show you the three ways you can load source files for debugging.

Note: For demonstration purposes, before you begin this section, perform the following from your shell:

```
% mkdir demos
% mkdir demos/jello
% cd demos/jello
% cp /usr/demos/WorkShop/jello/* .
% make
% cvd &
```

Load Directly into the Main View Window

Perform the following steps to load your source file directly into, or run your executable from, the Debugger Main View window:

- Enter the source file directly. Enter the name (or full pathname, if necessary) of the source file in the **File** field.

  For example, enter the following if you launched **`cvd &`** from within the `jello/demos` directory.:

  `File:` **`jello.c`**

- Enter the executable directly. Enter the name (or full pathname, if necessary) of the executable in the **Command** field.

For example, enter the following if you launched **cvd &** from within the `jello/demos` directory:

```
Command: jello
```

Load from the File Browser Dialog Box

You can load source files from the **File Browser** dialog box, available as **Views > File Browser** from the menu bar of the Main View window. The **File Browser** window is shown in Figure 3-1.

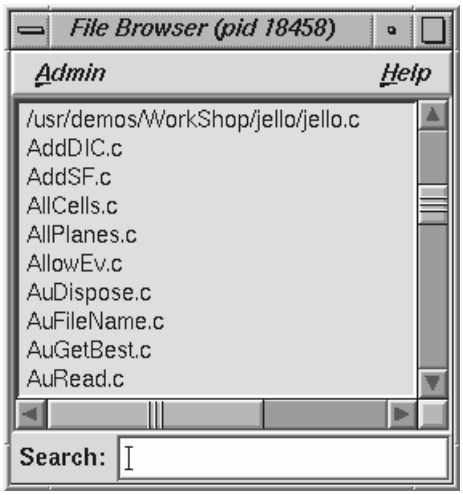

Figure 3-1 File Browser Window

This dialog box provides you with a list of the source files that your executable file can use, including any files in linked libraries.

- **locate a file**: to locate a file, enter your desired filename in the **Search** field.
- **load a file**: to load a file directly into the Main View window from the **File Browser** dialog box, simply double-click on the file name.

You may be unable to locate some files because the source supports system routines. Source for these routines may not be available on your system.

Load from the Open Dialog Box

You can load source files from the **Open** dialog box, available as **Source > Open** from the menu bar of the Main View window. The standard dialog box lists all available files and the currently selected directory in the **Selection** field. You can change this directory as you wish.

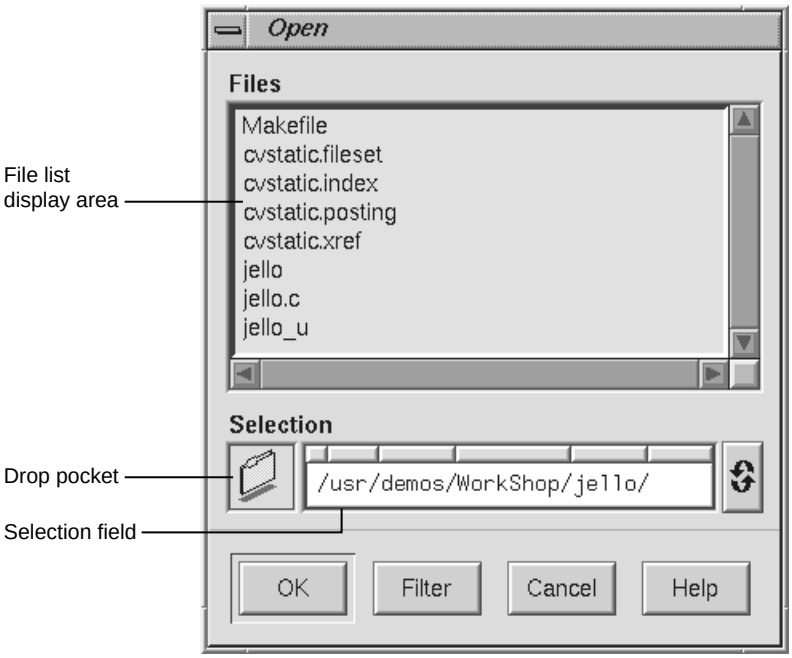

Figure 3-2 Open Dialog Box

There are several ways to load a file. You can:

- double-click on the file name.
- type the full pathname of the file in the **Selection** field and click the **OK** button.
- drag the file icon into the drop pocket. (Use an application like `fm` to produce file icons.)

If you specify a file name without a full path, the Debugger uses the current path remapping information to try to locate the file (see Path Remapping, "Path Remapping", page 38).

Path Remapping

Path remapping allows you to modify mappings to redirect file names, located in your executable file, to their actual source locations on your file system. Because WorkShop uses full (that is, absolute) path names, path remapping generally is not necessary. However, if you have mounted executable files on a different tree from the one on which they were compiled, you will need to remap the root prefix to get access to the source files in that hierarchy.

The most basic remapping is for ".", which allows you to specify the directories to be searched for files. This basic function works just like `dbx` and can be modified by using `use`/*full_path_name*(blank) and `dir`/*full_path_name*(blank) in the command line.

Open the **Path Remapping** window (**Admin > Remap Paths)** from the menu bar of the Main View window. The following window is displayed:

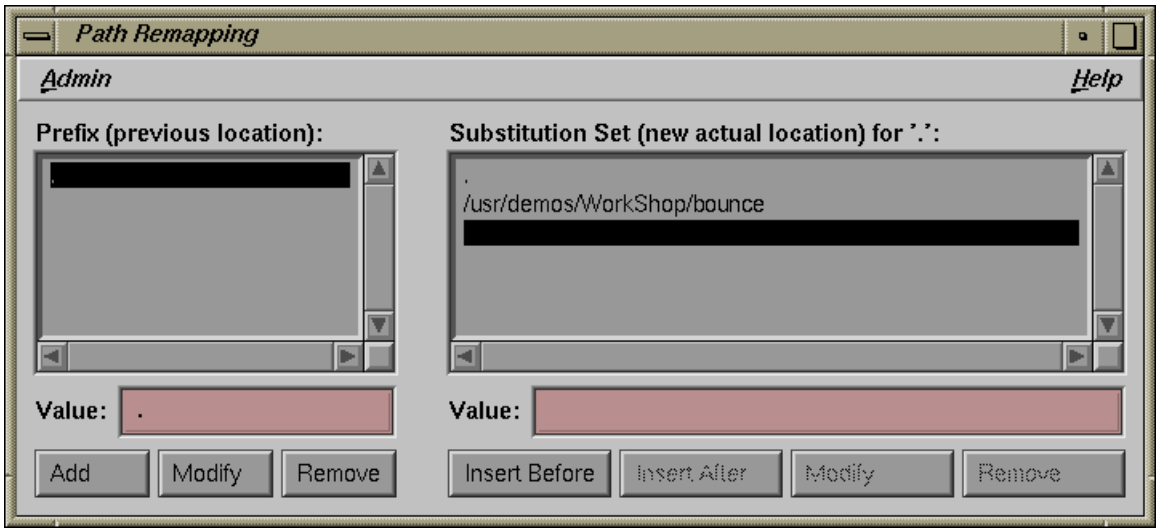

Figure 3-3 `Path Remapping` Dialog Box

For each prefix listed in the **Prefix** list, there is an ordered set of substitutions used to find a real file. By default, path remapping is initialized so that "." is mapped to the current directory. The **Substitution Set for '.'** list shows the substitution list for the currently highlighted item in the **Prefix** list. The **Prefix** list represents where the source file(s) used to be and the **Substitution Set** indicates where the source file(s) are currently. You can perform the following operations through the **Path Remapping** dialog box:

- To view the substitution set for a different prefix, click that prefix.
- To add a new prefix, enter the new value in the **Value** field below the **Prefix** list and click the **Add** button. A new substitution set is created with the prefix name as the first element. Click on this element to highlight it.

  Next, type the desired substitution in the **Value** field below the **Substitution Set** list and insert it by clicking on either the **Insert Before** button or the **Insert After** button.
- To modify the currently selected prefix, edit the string in the **Value** field and click the **Modify** button.
- To remove the current prefix and its substitution set, select the prefix and click the **Remove** button.

Case Example for Path Remapping

In some cases, if source files have been moved to new locations, path remapping is required to help the Debugger find the source files again.

The following tutorial shows you a case for remapping. It includes demo files bundled with your WorkShop Debugger:

1. Create a new directory in *your_home_directory*:

   ```
   % mkdir jellodemos
   ```

2. Change to the new directory:

   ```
   % cd jellodemos
   ```

3. Copy the Jello demo files from the Workshop demo directory into your new directory:

   ```
   % cp /usr/demos/WorkShop/jello/* .
   ```

4. Enter the following to ensure that the `jello` executable contains the `jello` demos source path:

   ```
   % make clobber
   % make
   ```

5. Create another new directory in your `jellodemos` directory:

   ```
   % mkdir ./newdir
   ```

6. Move the Jello source files to a new location:

   ```
   % mv ./*.c ../newdir
   ```

7. Start the WorkShop Debugger:

   ```
   % cvd ./jello &
   ```

   The Main View window will display with no source in the source pane. The following message appears:

   Unable to find file <*your_home_directory*/newdir/jello.c

8. Choose the following from the menu bar in the Main View window: **Admin > Remap Paths...**. The **Path Remapping** window displays.

9. In the **Substitution Set for '.':** dialog box:

   a. Select *your_home_directory*`/jellodemos/newdir/jello.c`

      The *path/filename* will appear in the **Value:** field.

   b. Enter the following:

      ```
      .
      ```

10. Enter the following in the **Value:** field below the **Substitution Set for '.':** dialog box:

    ```
    newdir
    ```

11. Click on the `Insert Before` button.

    The directory is inserted before the highlighted empty line in the **Substitution Set for '.':** dialog box and after the first element, which was not highlighted.

Now, the source appears in the Main View source pane as *your_home_directory*`/newdir/jello.c`

Tutorial: The `jello` Program

This chapter presents a short tutorial with the demonstration program `jello`, provided with your software. This tutorial walks you through commonly encountered debugging situations.

The chapter is divided into several parts:

Before you begin this tutorial, you should be aware of the following:

- This tutorial must be run on an SGI workstation.
- WorkShop identifies files with the path names in which they were compiled. The path names in the tutorial may not match the ones on your system.

Starting the Debugger

Use the following syntax to start the Debugger:

`cvd` [`-pid` *pid*] [`-host` *host*] [*executable_file* [*corefile*]] [`-args` *arg1 arg2 arg3 ...*][`&`]
The **`cvd`** command should be invoked in the same directory as your program source code.

- The `-pid` option lets you attach the Debugger to a running process. You can use this to determine why a live process is in a loop.
- The `-host` option lets you specify a remote host on which to run your program while the Debugger runs locally. This option is seldom used, except under the following circumstances:

  - You do not want the Debugger windows to interfere with the execution of your program on the remote host.
  - You are supporting an application remotely.
  - You do not want to use the Debugger on the target machine for any other reason.

  Note: The host and local machines must be running the same version of WorkShop. Also, the `.rhost` files on the machines must allow `rsh` commands to operate between them.

- The *executable_file* argument is the name of the executable file for the process or processes you want to run. This file is produced when you compile with the `-g` option, which disables optimization and produces the symbolic information necessary to debug with WorkShop. The `-g` option is most commonly used, but it is optional; if you wish, you can invoke the Debugger first and specify the name of the executable file later.
- Sometimes when a file is being executed, a core file is produced (its default name is `core`). Use the following command to determine why a program crashed and produced the core file:

  `cvd` *executable_file* `core`

- The `-args` option passes program arguments to the executable to be debugged.

See "Run the jello Program", page 42, for more information.

Run the `jello` Program

In this part of the tutorial, you invoke the Debugger and start a typical process running. The `jello` program simulates an elastic polyhedron bouncing around inside of a revolving cube. The program's functionality is mainly contained in a single loop that calculates the acceleration, velocity, and position of the polyhedron's vertices.

Enter the following commands to run the `jello` program:

1. Go to the directory with the `jello` demo program:

   > **`cd /usr/demos/WorkShop/jello`**

2. List the contents of this directory:

   >**`ls`**

3. Enter the following to make the program if `jello` is not listed (from Step 2):

   > **`make jello`**

4. Invoke the Debugger with the `jello` program:

   > **`cvd jello &`**

   The Main View window appears, as shown in Figure 4-1, page 44, and scrolls automatically to the `main` function.

   In addition to the Main View window, the **Execution View** icon also appears. When you run the `jello` program, the command you used to invoke `jello` is displayed in this window.

   The **Execution View** window is the interface between the program and the user for programs that use standard input/output or generate `stderr` messages.

   The Main View window brings up the source file in read-only mode. You can change this to read/write mode if you select **Source > Make Editable** from the Main View window menu bar (provided you have the proper file access permissions).

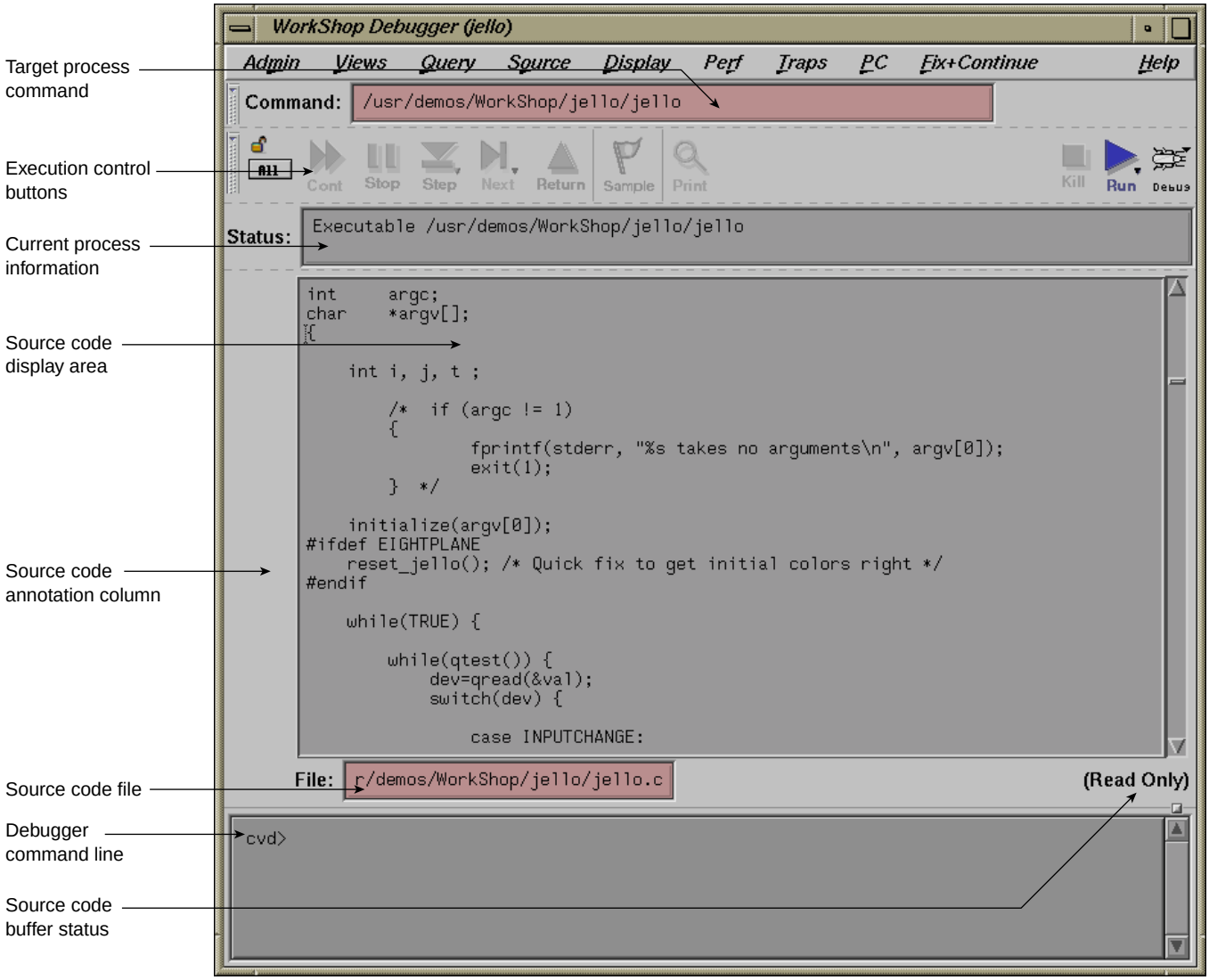

Figure 4-1 The Main View Window with `jello` Source Code

5. Click the **Run** button in the upper-right corner of the Main View window to run the `jello` program.

The `jello` window opens on your display (see Figure 4-2, page 45). Enlarge this window to watch the program execute. The polyhedron is initially suspended in the center of the cube.

If you wish, you can perform the following with the `jello` program:

a. Click the left mouse button anywhere inside the `jello` window.

   The polyhedron drops to the floor of the cube.

b. Hold down the right mouse button to display the pop-up menu and select **spin**.

   The cube rotates and the polyhedron bounces inside the cube.

c. Hold down the right mouse button to display the pop-up menu and select the **display** option.

   This opens a submenu that allows you to change the appearance of the `jello` polyhedron.

d. Feel free to select (right-click) from this menu to see how the `jello` display changes.You may encounter flashing colors inside windows while running `jello`. This is normal.

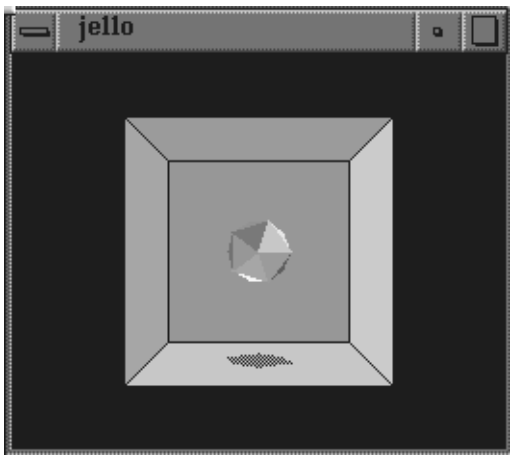

Figure 4-2 The `jello` Window

6. Right-click and select **exit** from the pop-up menu to exit this demonstration executable.

Perform a Search

This part of the tutorial covers the search facility in the Debugger. You will search through the `jello` source file for a function called `spin`. The `spin` function recalculates the position of the cube.

1. Select **Source > Search** from the Main View menu bar.

   The **Search** dialog box appears.

2. Type **`spin`** in the **Search** field in the dialog box, as shown in Figure 4-3.

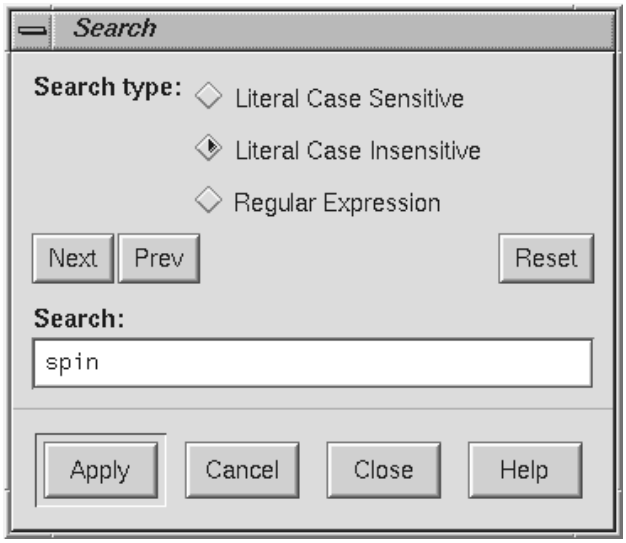

Figure 4-3 The **Search** Dialog

3. Click the **Apply** button.

   The search takes place on the displayed source files. Each instance of `spin` is highlighted in the source code and flagged with target indicators in the scroll bar to the right of the display area. (See search target indicators in Figure 4-4, page

48.) The **Next** and **Prev** buttons in the **Search** dialog box let you move from one occurrence to the next in the order indicated.

For more information on **Search**, see "Source Menu", page 183.

4. Click the **Close** button and the dialog box disappears.
5. Click the middle mouse button on the last search target indicator at the right side of the source code pane (see the figure below). This scrolls the source code down to the last occurrence of `spin`, the location of the spin function.

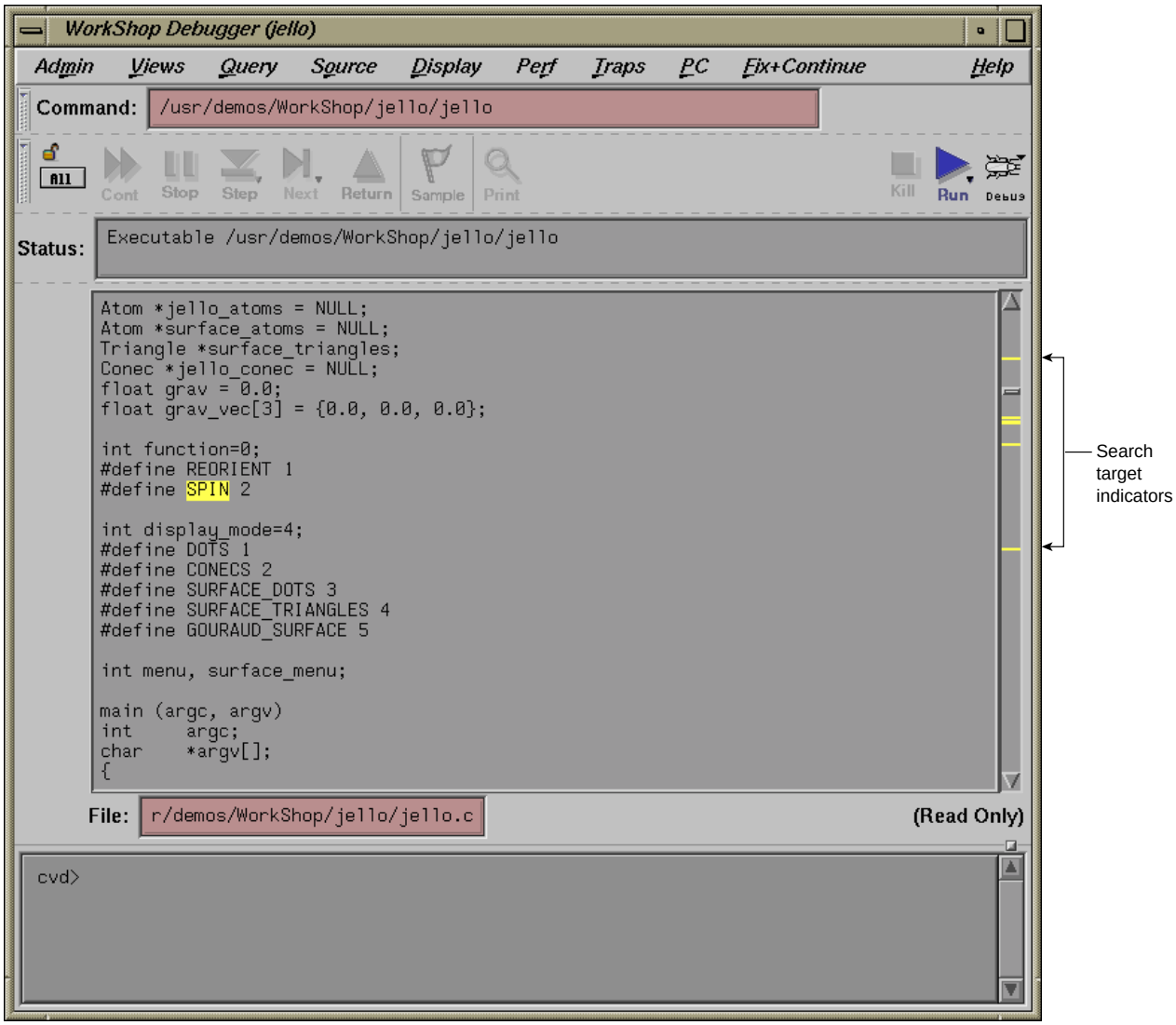

Figure 4-4 Search Target Indicators

6. Proceed to "Edit Your Source Code" to edit your code.

Edit Your Source Code

To edit and recompile your source code, follow these steps:

1. Select **Source > Fork Editor** from the Main View menu bar. A text editor appears. (In this case, if you are proceeding from "Perform a Search", page 46, notice that the **spin** function is displayed.)

   If you use source control, you can check out the source code through the configuration management shell by selecting **Source > Versioning > CheckOut** from the menu bar.

2. Edit the source code as follows:

   Change all occurrences of `3600` to `3000` in the following code:

   ```
   if ((a+=1)>3600) a -= 3600;
   if ((b+=3)>3600) b -= 3600;
   if ((c+=7)>3600) c -= 3600;
   ```

   Save your changes.

3. Select **Source > Recompile** from the menu bar to recompile your code.

   The **Build View** window displays and starts the compile. Your `makefile` will determine which files need to be recompiled and linked to form a new executable.

   Any compile errors are listed in the window, and you can access the related source code by clicking the errors. This does not apply to warnings generated by the compiler.

   For more information on compiling, see Appendix B, "Using the Build Manager", page 329.

   When the code is successfully rebuilt, the new executable file reattaches automatically to the Debugger and the Static Analyzer. Previously set traps are intact unless you have traps triggered at line numbers and have changed the line count.

Setting Traps

Stop traps (also called breakpoints and watchpoints) stop program execution at a specified line in the code. This allows you to track the progress of your program and to check the values of variables at that point. Typically, you set breakpoints in your

program prior to running it under the Debugger. For more information on traps, see to Chapter 5, "Setting Traps", page 65.

In this part of the tutorial, you set a breakpoint at the `spin` function.

1. Click the **Run** button to run the `jello` executable.

   The demo window will display.

2. Click the left mouse button in the Main View source code annotation column next to the line containing `if ((a+=1)>3600) a -= 3600;`, or `if ((a+=1)>3000) a -= 3000;`, if you are proceeding from the previous section).

   A stop trap indicator appears in the annotation column as shown in Figure 4-5, page 51.

3. Right-click on the `jello` window and select `spin` from the pop-up menu.

   The program runs up to your stop trap and halts at the beginning of the next call to the `spin` function. When the process stops, an icon appears and the line is highlighted. Note that the `Status` field indicates the line at which the stop occurs.

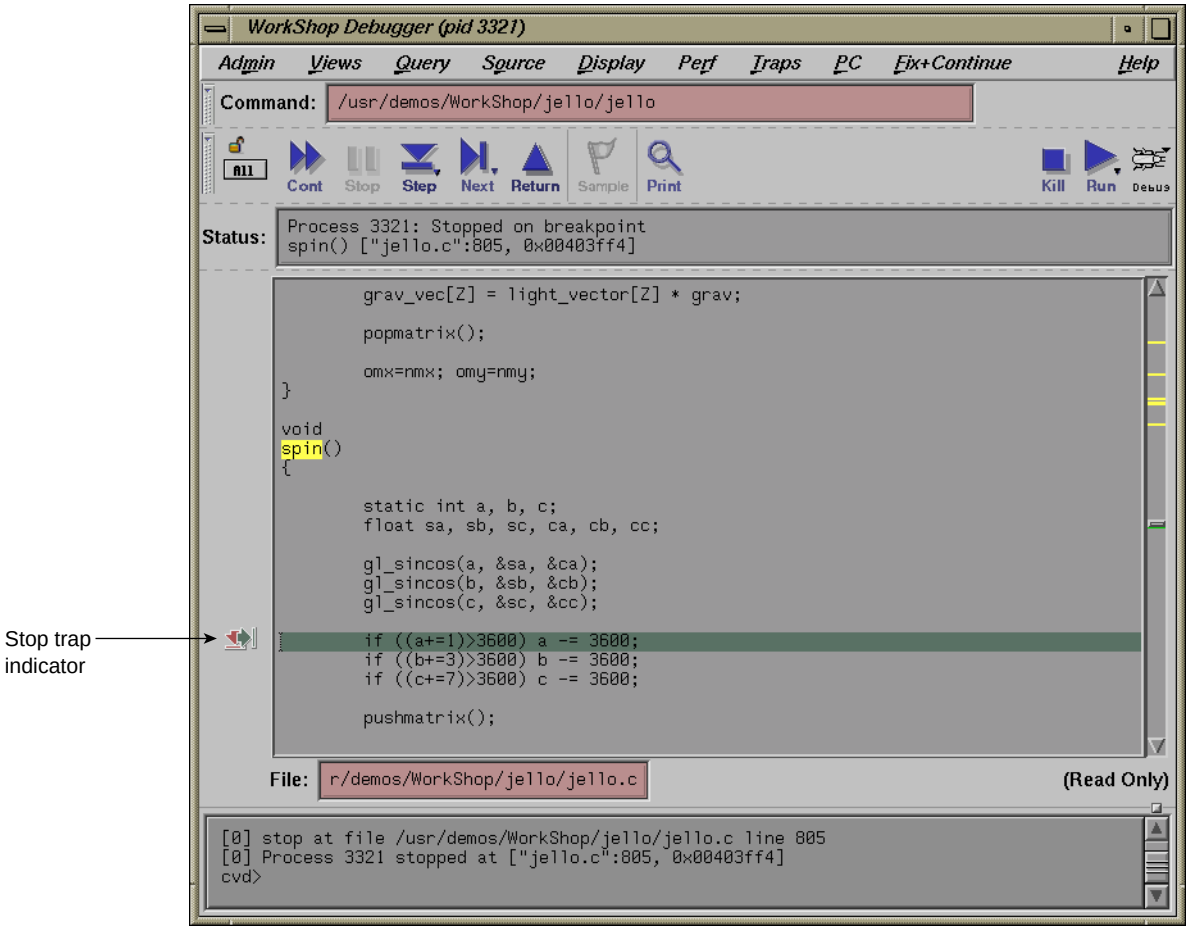

Figure 4-5 Stop Trap Indicator

4. Click the **Continue** button at the upper-left corner of the Main View window several times. Observe that the `jello` window goes through a spin increment with each click.

5. Select **Views > Trap Manager** from the Main View window menu bar.

   The **Trap Manager** window appears as shown in Figure 4-6, page 53.

The **Trap Manager** window lets you list, add, edit, disable, or remove traps in a process. In Step 2, you set a breakpoint in the `spin` function by clicking in the source code annotation column. The trap now displays in the trap display area of the **Trap Manager** window.

The **Trap Manager** window also lets you to do the following:

- Define other traps.
- Set conditional traps in the **Condition** field near the top of the window.
- Specify the number of times a trap should be encountered before it activates by using the **Cycle Count** field.
- Manipulate traps by using trap controls (**Modify**, **Add**, **Clear**, **Delete**).
- View all traps (active and inactive) in the trap display area.

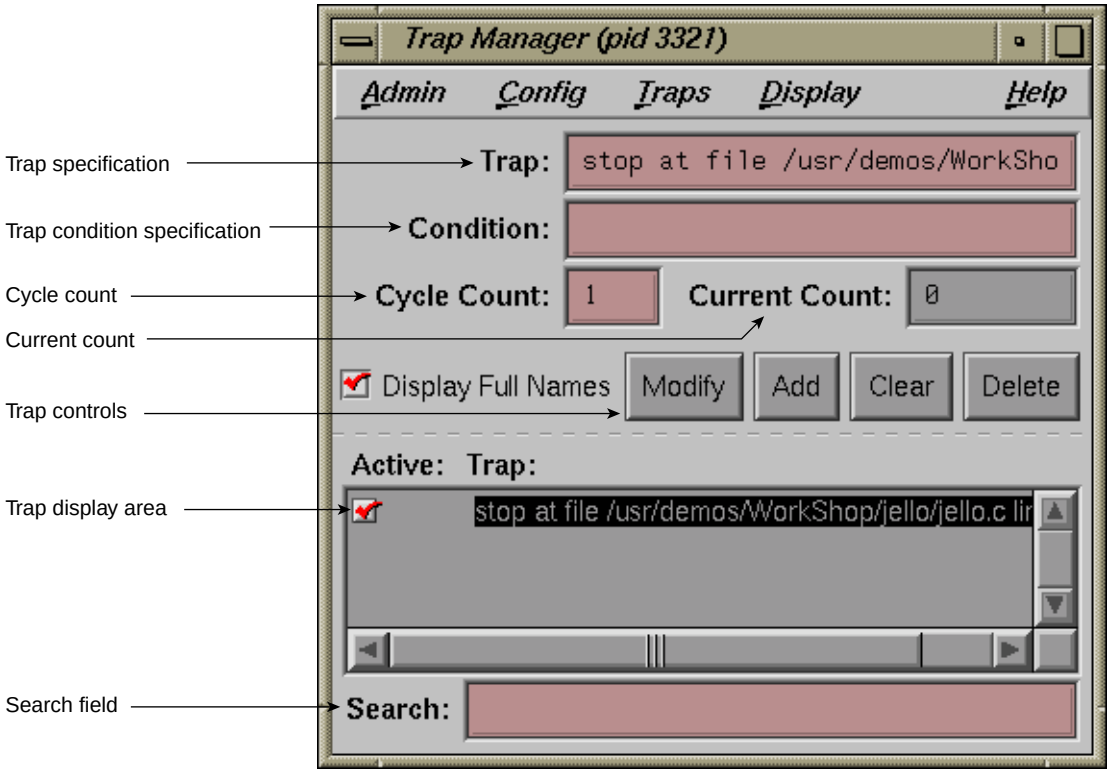

Figure 4-6 Trap Manager Window

6. Click the button to the left of the stop trap in the trap display area.

   The trap is temporarily disabled. (If you click again in this box, the trap will be re-enabled.)

7. Click the **Clear** button, move the cursor to the **Trap** field, then type:

   ```
   watch display_mode
   ```

   Click the **Add** button.

   This sets a watchpoint for the `display_mode` variable. A *watchpoint* is a trap that causes an interrupt when a specified variable or address is read, written, or executed.

8. Click the **Cont** button in the Main View window to restart the process.

   The process now runs somewhat slower but still at a reasonable speed for debugging.

9. Hold down the right mouse button in the `jello` window to display the pop-up menu. From this menu, select **display** and then select the **conecs** option with the right mouse button.

   This triggers the watchpoint and stops the process.

10. Go to the **Trap Manager** window and click the button next to the `display_mode` watchpoint to deactivate it. Then, click the button next to the `spin` stop trap to reactivate it.

11. Enter **`100`** in the **Cycle Count** field and click the **Modify** button. Notice how the trap description changes in the **Trap Manager** window.

12. Click the **Continue** button in the Main View window.

    This takes the process through the stop trap for the specified number of times (100), provided no other interruptions occur.

    The **Current Count** field keeps track of the actual number of iterations since the last stop, which is useful if an interrupt occurs. Note that it updates at interrupts only.

13. Select **Close** from the **Admin** menu to close the **Trap Manager** window.

Examining Data

This part of the tutorial describes how to examine data after the process stops.

1. Select **Views > Call Stack** from the Main View window menu bar.

   The **Call Stack** window appears as shown in Figure 4-7, page 55. The **Call Stack** window shows each frame in the call stack at the time of the breakpoint, with the calling parameters and their values. Through the **Call Stack > Display** menu, you can also display the calling parameters' types and locations, as well as the program counter (PC) . The program counter is the address at which the program has stopped. For more information about the program counter, see "Traceback Through the Call Stack Window", page 85.

In this example, the `spin` and `main` stack frames are displayed in the **Call Stack** window, and the `spin` stack frame is highlighted, indicating that it is the current stack frame.

2. Select **Admin > Active** from the **Call Stack** window menu bar.

   Notice that the **Active** toggle button is turned on. Active views are those that have been specified to change their contents at stops or at call stack context changes. If the toggle is on, the call stack is updated automatically whenever the process stops.

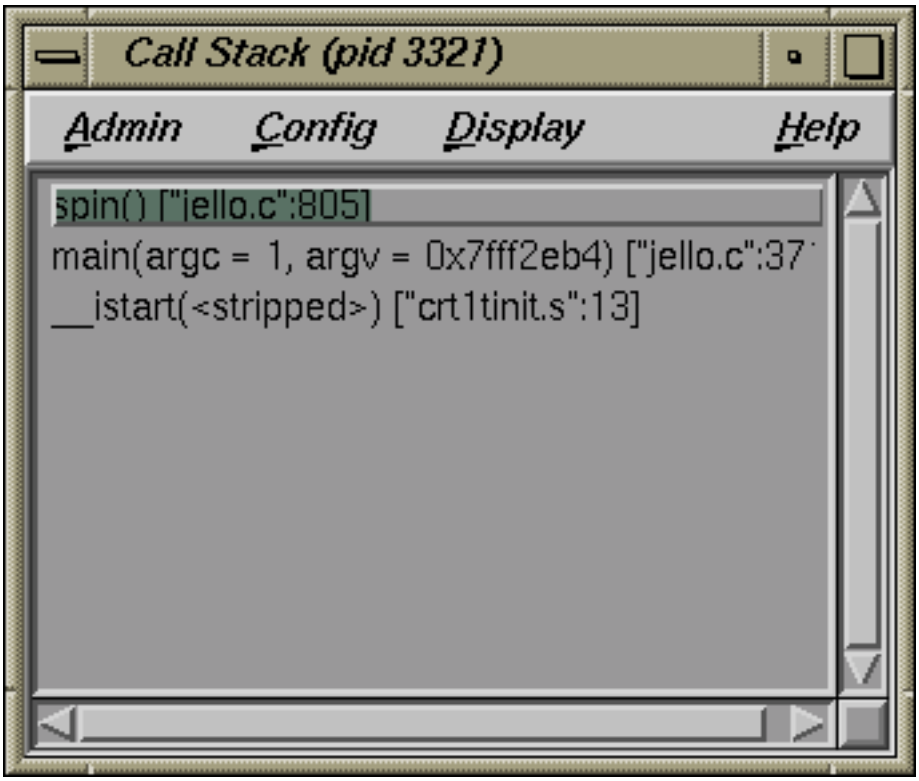

Figure 4-7 Call Stack at `spin` Stop Trap

3. Double-click the `main` stack frame.

This shifts the stack frame to the `main` function, scrolls the source code in the Main View window (or **Source View**) to the place in `main` where `spin` was called, and highlights the call. Any active views are updated according to the new stack frame.

4. Double-click the `spin` stack frame.

   This returns the stack frame to the `spin` function.

   Select **Variable Browser** from the **Views** menu in the Main View window.

   The **Variable Browser** window appears. This window shows you the value of local variables at the breakpoint. The variables appear in the left column (read-only), and the corresponding values appear in the right column (editable).

   Your **Variable Browser** window should resemble the one in Figure 4-8, although you may need to enlarge the window to see all the variables (the values will be different).

   The `jello` program uses variables *a, b,* and *c* as angles (in tenths); *ca, cb, cc* as their corresponding cosines; and *sa, sb, sc* as their sines. Whenever you stop at `spin`, these values change.

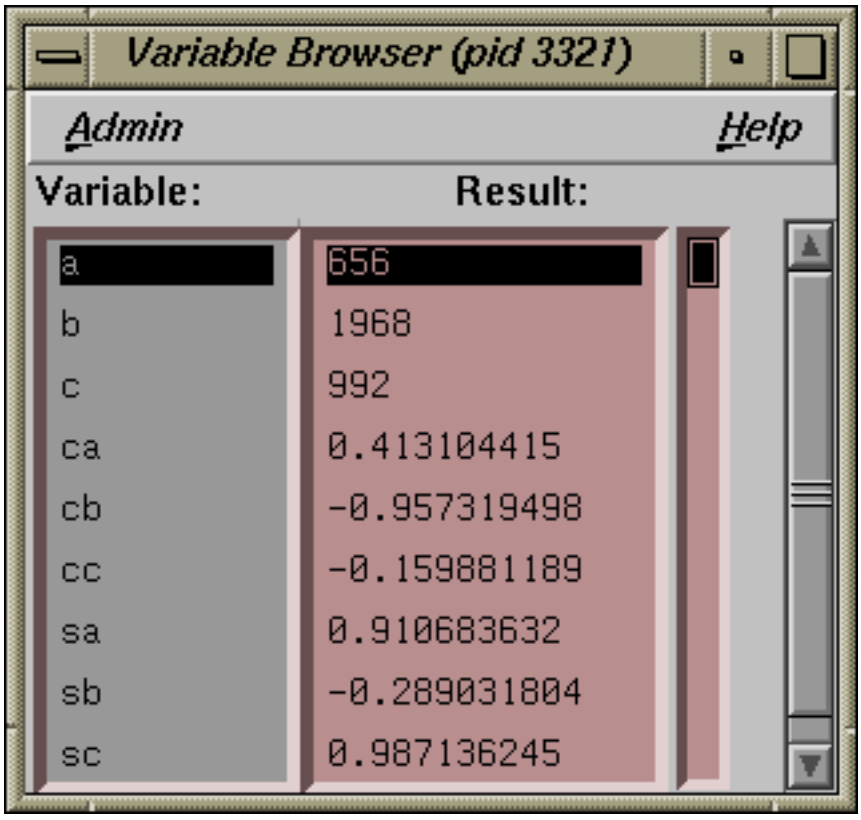

Figure 4-8 Variable Browser at `spin`

5. Double-click some different frames in **Call Stack** and observe the changes to **Variable Browser** and the Main View window.

   These views update appropriately whenever you change frames in **Call Stack**. Notice also the change indicators in the upper-right corners of the **Result** fields in Figure 4-9, page 58. These appear if the value has changed. If you click the *folded* corner, the previous value displays (and the indicator appears *unfolded*). You can then toggle back to the current value.

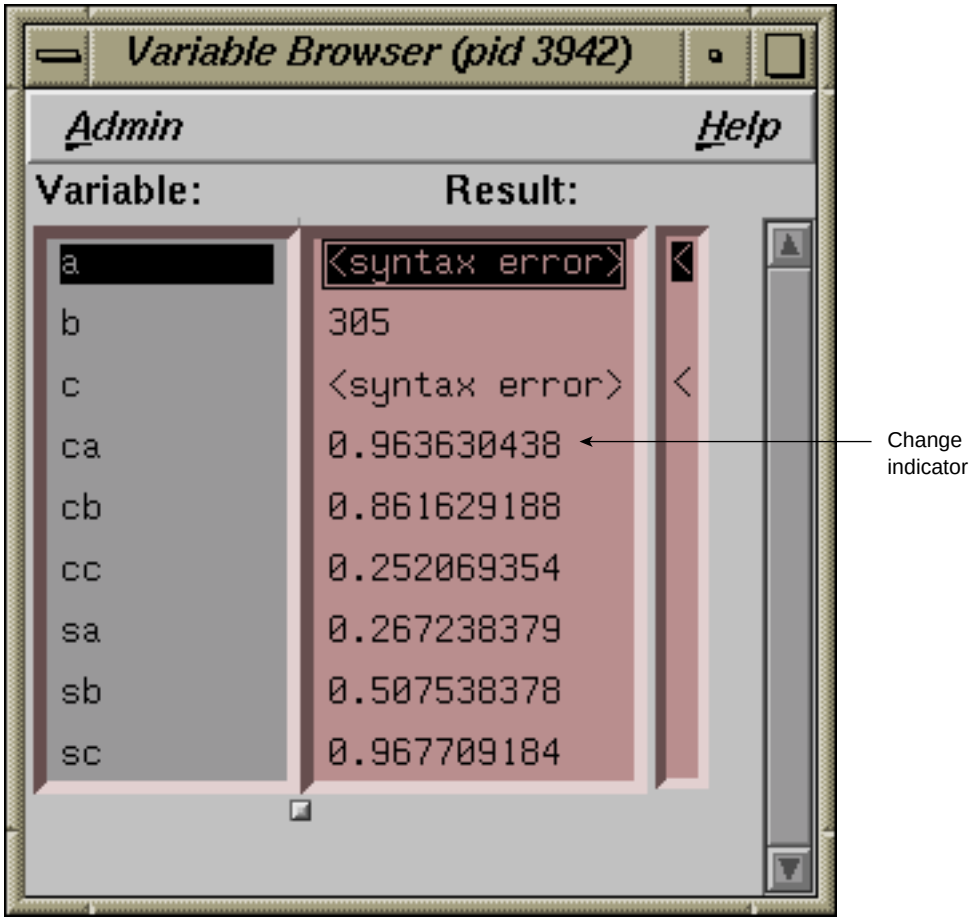

Figure 4-9 Variable Browser after Changes

6. Select **Close** from the **Admin** menu in **Variable Browser** and **Close** from the **Admin** menu in **Call Stack** to close them.
7. Select **Expression View** from the **Views** menu in the Main View window.

   The **Expression View** window appears. It lets you evaluate an expression involving data from the process. The expression can be typed in, or more simply, cut and pasted from your source code. You can view the value of variables (or expressions involving variables) any time the process stops. Enter the expression

in the left column, and the corresponding value appears in the right column. For more information, see "Evaluating Expressions", page 91.

8. Hold down the right mouse button in the **Expression** column to bring up the **Language** menu. Then hold down the right mouse button in the **Result** column to display the **Format** menu.

   The **Language** menu (shown on the left side of Figure 4-10, page 59) lets you apply the language semantics to the expression.

   The **Format** menu (shown on the right side of Figure 4-10, page 59) lets you view the value, type, address, or size of the result. You can further specify the display format for the value and address.

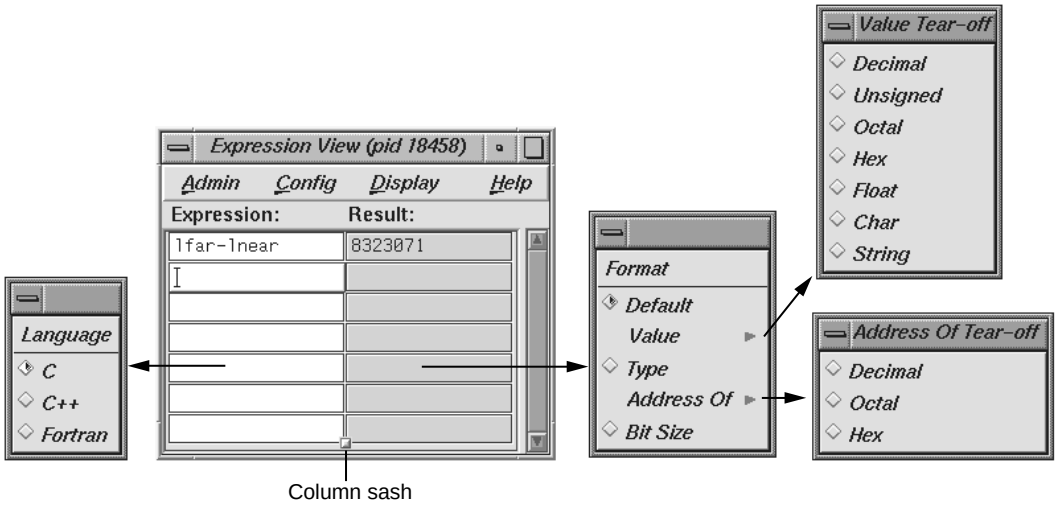

Figure 4-10 Expression View with **Language** and **Format** Menus Displayed

9. Click on the first **Expression** field in the **Expression View** window. Then enter **`(a+1)>3600`** in the field and press `Enter`.

   This is a test performed in `jello` to ensure that the value of *a* is less than 3600. This uses the variable *a* that was displayed previously in **Variable Browser**. After you press `Enter`, the result is displayed in the right column; `0` signifies FALSE.

10. Select **Admin > Close** from the **Expression View** window menu bar to close that window.

11. Select **Views > Structure Browser** from the Main View window to open the **Structure Browser**.

12. Enter **jello_conec** in the **Expression** field and press Enter.

    The **Structure Browser** window displays the structure for the given expression; field names are displayed in the left column, and values in the right column. If only pointers are available, the **Structure Browser** will de-reference the pointers automatically until actual values are encountered. You can then perform any further de-referencing by double-clicking pointer addresses in the right column of the data structure objects. A window similar to the one shown in Figure 4-11 now appears.

Figure 4-11 Structure Browser Window with jello_conec Structure

13. Click once to focus, then double-click the address of the **next** field (in the right column of the jello_conec structure).

    Double-clicking the address corresponding to a pointer field de-references it. Double-clicking the field name displays the complete name of the field in the **Expression** field at the top of the **Structure Browser** window. (See Figure 4-12, page 61.)

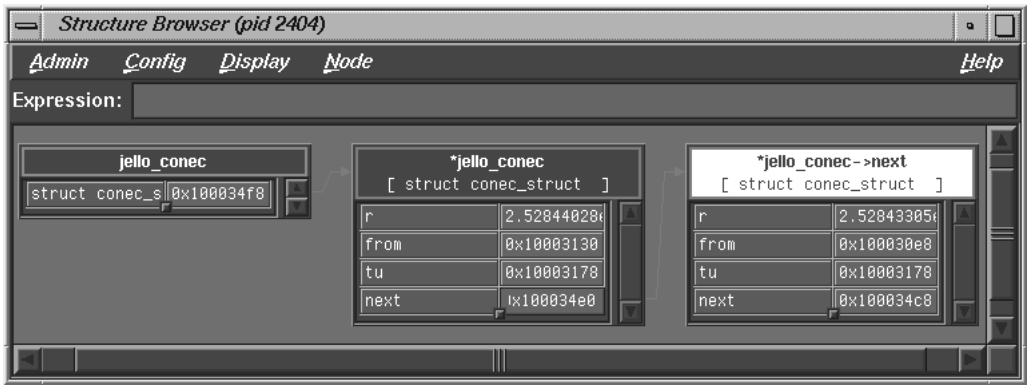

Figure 4-12 Structure Browser Window with `Next` Pointer De-referenced

14. Select **Close** from the **Admin** menu of **Structure Browser** window to close it.
15. Select **Views > Array Browser** from the Main View window menu bar.

    The **Array Browser** lets you see or change values in an array variable. It is particularly valuable for finding bad data in an array or for testing the effects of values you enter.
16. Type **`shadow`** in the **Array** field and press `Enter`.

    You can now see the values of the `shadow` matrix, which displays the polyhedron's shadow on the cube. The **Array Browser** template should resemble Figure 4-13, page 62, but with different data values. If any areas are hidden, hold down the left mouse button and drag the sash buttons at the lower right of the array specification and subscript control areas to expose the area.

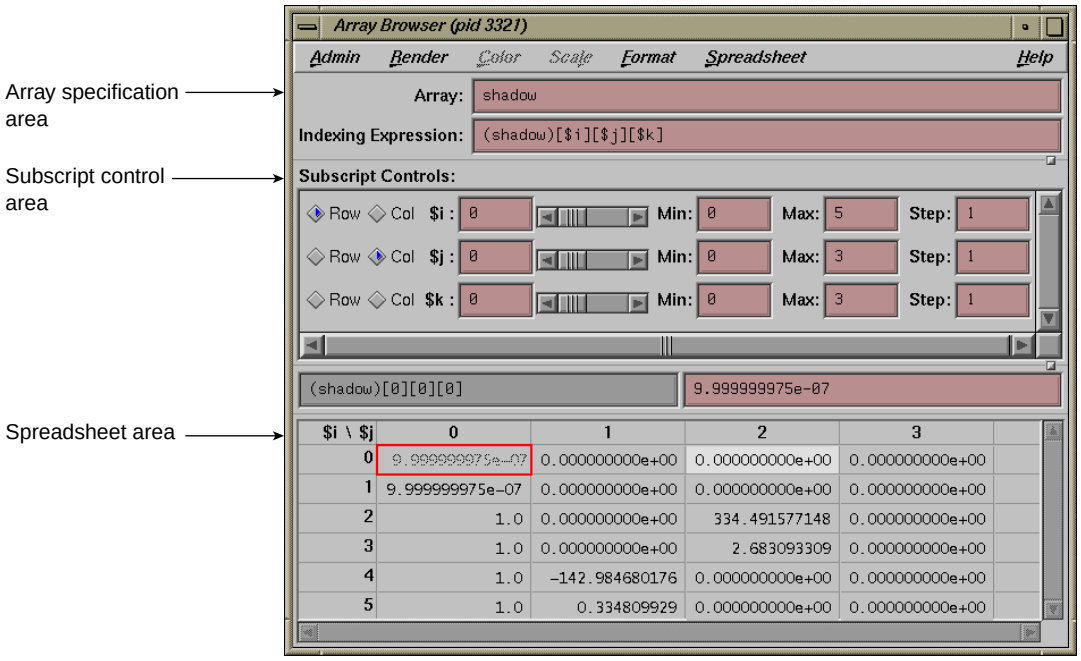

Figure 4-13 Array Browser Window for `shadow` Matrix

17. Select the **Col** button next to the `$k` index in the **Subscript Controls** pane (you may need to scroll down to it).

    The **Array Browser** can handle matrices containing up to six dimensions but displays only two dimensions at a time. Selecting the **Col** button for `$k` has the effect of switching from a display of `$i` by `$j` to a display of `$i` by `$k`.

    Figure 4-14, page 63, shows a close-up view of the subscript control area.

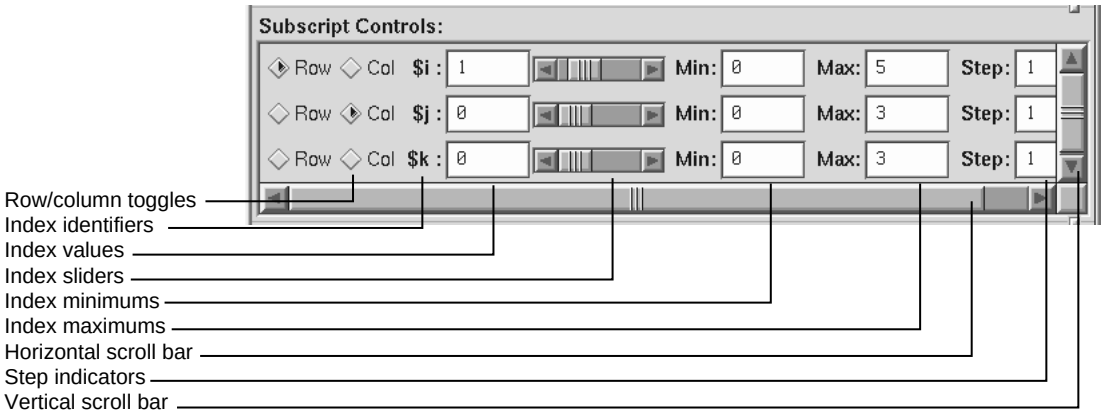

Figure 4-14 `Subscript Controls` Panel in **Array Browser** Window

The row and column toggles indicate whether a particular dimension of the array appears as a row, column, or not at all in the spreadsheet area. Although an array may be of 1 to 6 dimensions, you can view only one or two dimensions at a time. The index values shown as **Min** and **Max** initially exhibit the lower and upper bounds of the dimension indicated. The values may be changed to allow the user to display a subset of the available index range in that dimension. The index sliders let you move the focus cell along the particular dimension. The focus cell may also be changed by selecting a cell with the left mouse button. The index slider for a dimension whose row and column toggles are both off may be used to select a different two–dimensional plane of a multidimensional array. Use the horizontal and vertical scroll bars to expose hidden portions of the **Array Browser** window.

18. Select **Surface** from the **Render** menu.

    The **Render** menu displays the data from the selected array variable graphically, in this case as a three-dimensional surface. The selected cell is highlighted by a rectangular prism. The selected subscripts correspond to the x- and y-axes in the rendering with the corresponding value plotted on the z-axis. The data can be rendered as a surface, bar chart, multiple lines, or points.

Exiting the Debugger

There are several ways to exit the Debugger:

- Select **Exit** from the **Admin** menu.
- Type **quit** at the Debugger command line as follows:

  ```
  cvd> quit
  ```

- Double-click on the icon in the upper-left corner of the Main View window.
- Press Ctrl-c in the same window where you entered the **cvd** command.

Setting Traps

Traps are also referred to as breakpoints, watchpoints, samples, signals, and system calls. Setting traps is one of the most valuable functions of a debugger or performance analyzer. A trap enables you to select a location or condition within your program at which you can stop the execution of the process, or collect performance data and continue execution. You can set or clear traps from the Main View window or the **Trap Manager** window. You can also specify traps in the Debugger command line at the bottom of the Main View window. For signal traps, you can also use the **Signal Panel** window; and for system call traps, use the **Syscall Panel** window.

When you are debugging a program, you typically set a trap in your program to determine if there is a problem at that point. The Debugger lets you inspect the call stack, examine variable values, or perform other procedures to get information about the state of your program.

Traps are also useful for analyzing program performance. They let you collect performance data at the selected point in your program. Program execution continues after the data is collected.

This chapter covers the following topics:

For a tutorial on the use of traps, see "Setting Traps", page 49.

Traps Terminology

A *trap* is an intentional process interruption that can either stop a process or capture data about a process. It has two parts: the *trigger* that specifies when the trap fires and the *action* that specifies what happens when the trap fires.

Trap Triggers

You can set traps at a specified location in your program or when a specified event occurs. You can set a trigger at any of the following points:

- At a given line in a file (often referred to as a *breakpoint*)
- At a given instruction address
- At the entry or exit for a given function
- After set time intervals (referred to as a *pollpoint*)
- When a given variable or address is read, written, or executed (referred to as a *watchpoint*)
- When a given signal is received
- When a given system call is entered or exited

In addition, you can use an expression to specify a condition that must be met before a trap fires. You can also specify a *cycle count*, which specifies the number of passes through a trap before firing it.

When you set a breakpoint in C++ code that uses templates, that breakpoint will be set in each instantiation of the template.

Trap Types

Traps can affect any of a number of actions in your debugging process. The following is a list of the variety of traps and their functions. See "Syntaxes", page 74 for trap syntaxes.

- A *stop trap* will cause one or all processes to stop. In single process debugging, a stop trap stops the current process. In multiprocess debugging, you can specify a stop trap to stop all processes or only the current process.
- A *pending trap* is a trap with a destination address that does not resolve at the initial startup of the debugger: it allows you to place breakpoints on names that do not yet exist. By enabling pending traps, you make unknown function names acceptable breakpoint locations or entries. Because of this, the trap manager will not error a pending trap off if it can not be found in the executable. Instead, when a dlopen of a DSO (dynamic shared object) occurs, and the function name is found that matches the trap, the trap is resolved and becomes active.

There are two ways to enable pending traps.

– You can add the following line to your `.Xdefaults` file:

```
*AllowPendingTraps: true
```

– You can enable pending traps by entering the following in the `cvd>` pane of the Main View window:

```
set $pendingtraps=true
```

The default for this variable is `false`.

Only "stop entry" / "stop in" and "stop exit" traps can be pending traps, not "stop at" traps. The debugger will only accept a "stop at" trap on a DSO (dynamic shared object) and line number loaded at startup. (See "Syntaxes", page 74 for trap syntaxes.)

To get around this, set a pending "stop in" trap and run to the trap location. Now you can set "stop at" traps because the DSO is loaded. The debugger will then remember these traps if you run, kill, and re-run *within a single session*.

Pending traps display with the prefix `(pending)` preceding the normal trap display line.

Breakpoints on misspelled names are not flagged as errors when pending traps are enabled. This is because the Debugger has no way of knowing if a pending trap will ever be tripped.

- A *sample trap* collects performance data. Sample traps are used only in performance analysis, not in debugging. They collect data without stopping the process. You can specify sample traps to collect such information as call stack data, function counts, basic block counts, PC profile counts, `mallocs/frees`, system calls, and page faults. Sample traps can use any of the triggers that stop traps use. Sample traps are often set up as pollpoints so that they collect data at set time intervals.
- An *exception trap* fires when a C++ exception is raised.

  You can add a conditional expression to an exception trap through the **Trap Manager** window. However, the context in which the expression is evaluated is not that of the throw; the context is the exception handling of the C++ runtime library. Therefore, only global variables have unambiguous interpretation in the `if` clause.

  You should not include complex expressions involving operators such as `*` and `&` in your type specification for an exception trap. If you create an exception trap

with a specific base type, however, you will also stop your program on throws of pointer, reference, const, and volatile types. For example, if you use the following, your program will stop at type `char`, `char*`, `char&`, `const char&`, and so forth.:

```
cvd> stop exception char
```

Setting Traps

You can set traps directly in the Main View window by using the **Traps** menu or by clicking the mouse in the source annotation column. You can also specify traps at the Debugger `cvd` command line.

The following sections describe the ways that traps can be set:

- "Setting Traps with the Mouse", page 68
- "Setting Traps Using the cvd Command Line", page 68
- "Setting Traps Using the Traps Menu in the Main View Window", page 69

Setting Traps with the Mouse

The following lists ways to set traps by using your mouse:

- The quickest way to set a trap is to click in the source annotation column in the Main View or **Source View** windows. A subsequent click removes the trap.
- If data collection mode has been specified in the **Performance Data** window, clicking produces a sample trap; otherwise, a stop trap is entered.

  To determine if data collection is on, look at the upper-right corner of the Main View window to see which debugging option is selected (**Debug Only**, **Performance**, or **Purify**).

When the trap is set, a trap icon appears.

Setting Traps Using the cvd Command Line

The `cvd` command line is discussed in the `cvd`(1) man page. `stop` commands similar to those found in `dbx`, or as documented in "Setting Single-Process and Multiprocess Traps", page 72, may be entered into the command line portion of the Main View window as an alternative way to set traps.

Setting Traps Using the Traps Menu in the Main View Window

To set a trap using the **Traps** menu, you first need to know which type of trap you wish to set, then select the location in your program at which to set the trap.

To set a stop trap or sample trap at a line displayed in the Main View window (or the **Source View** window), click in the source code display area next to the desired line in the source code, or click-drag to highlight the line. Then, select either **Traps > Set Trap > Stop** or **Traps > Set Trap > Sample** from the menu bar.

The **Traps** menu in the Main View window is shown in Figure 5-1.

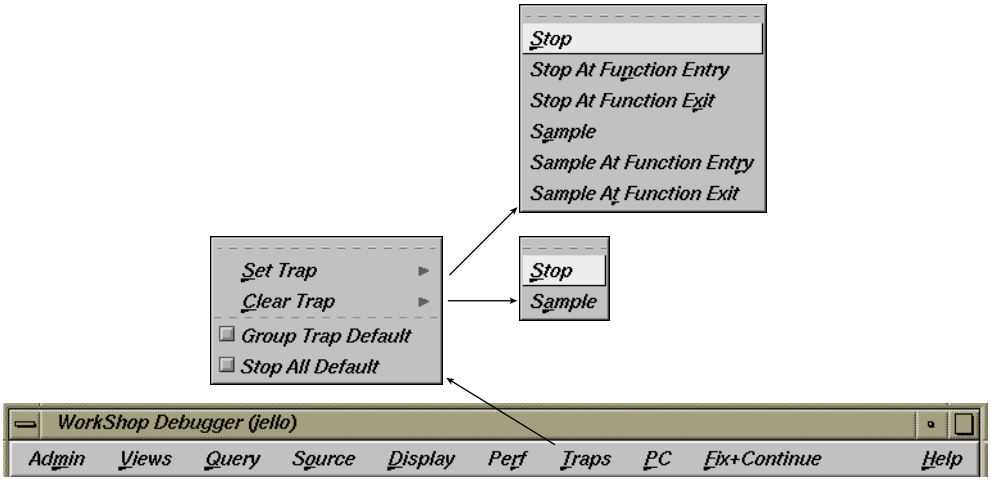

Figure 5-1 Traps Menu in the Main View Window

For a trap at the beginning or end of a function, highlight the function name in the source code display area and select one of the following from the **Traps > Set Traps** submenu as is appropriate to your needs:

- **Stop At Function Entry**
- **Stop At Function Exit**
- **Sample At Function Entry**
- **Sample At Function Exit**

Traps are indicated by icons in the source annotation column (and also appear in the **Trap Manager** window if you have it open). Figure 5-2, page 71, shows some typical trap icons. Sampling is indicated by a dot in the center of the icon. Traps appear in normal color or grayed out, depending on whether they are active or inactive. A transcript of the trap activity appears in the Debugger command line area. The active/inactive nature of traps is discussed in "Enabling and Disabling Traps", page 79.

The **Clear Trap** selection in the **Traps** menu deletes the trap on the line containing the cursor. You must designate a **Stop** or **Sample** trap type, since both types can exist at the same location appearing superimposed on each other.

When the **Group Trap Default** toggle is checked (ON), the `pgrp` option is added into the resulting trap when a trap is set. This option causes the trap to apply to all processes/pthreads in the group of which the current process is a member.

When the **Stop All Defaults** toggle is checked (ON), the `all` option is added into the resulting trap when a trap is set. This option causes the trap to apply to all processes/pthreads in the current debugging session.

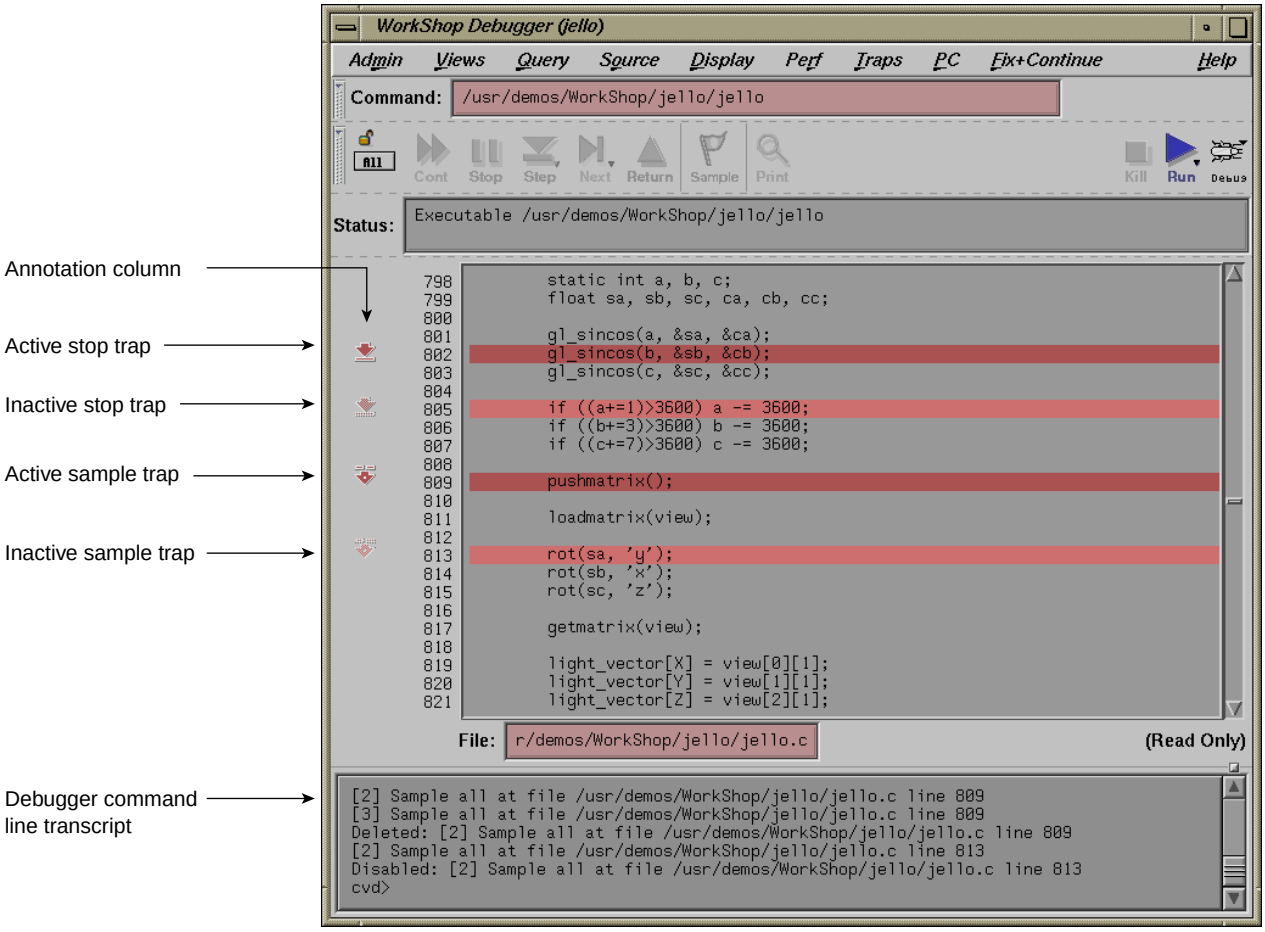

Figure 5-2 Typical Trap Icons

Setting Traps in the Trap Manager Window

The **Trap Manager** window is brought up by selecting **Views > Trap Manager** from the Main View window menu bar. This tool helps you manage all traps in a process. Its two major functions are to:

- List all traps in the process (except signal traps)

- Add, delete, modify, or disable the traps listed

The **Trap Manager** window appears in Figure 5-3 with the **Config**, **Traps**, and **Display** menus shown.

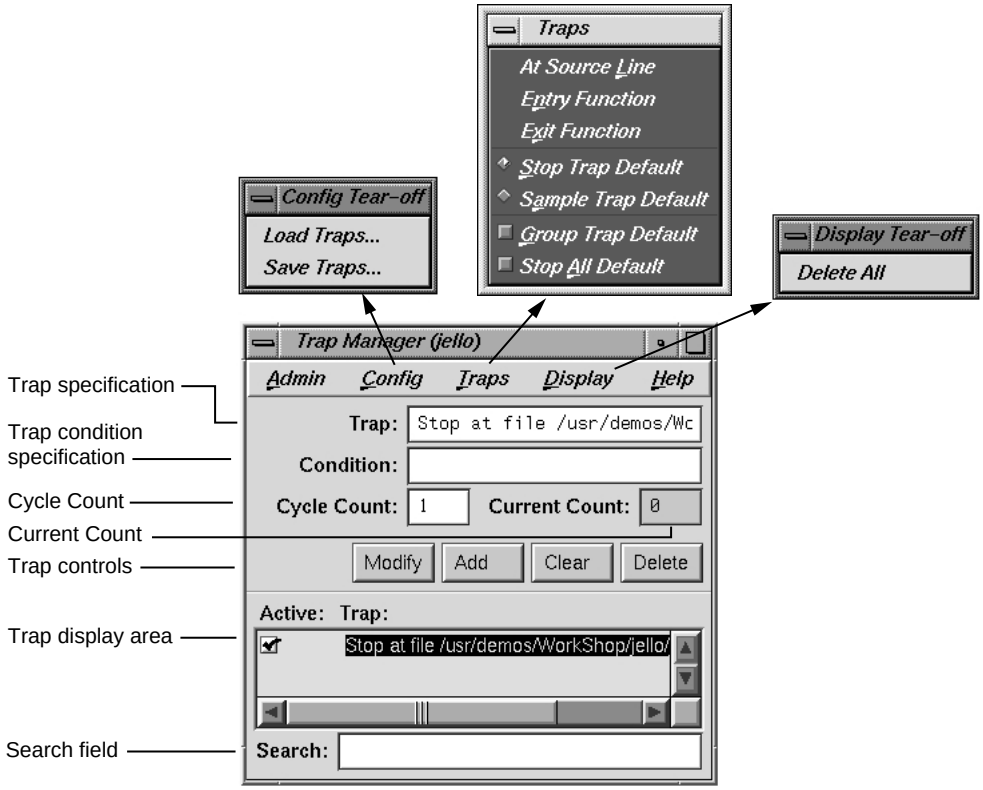

Figure 5-3 Trap Manager **Config**, **Traps**, and **Display** Menus

Setting Single-Process and Multiprocess Traps

New or modified traps are entered in the **Trap** field. Traps have the following general form:

```
[stop|sample] [all] [pgrp] location | condition
```

The [stop|sample] option refers to the trap action. You can set a default for the action by using the **Stop Trap Default** or **Sample Trap Default** selections of the **Traps** menu and omitting it on the command line.

The [all] and [pgrp] options are used in multiprocess analysis. The [all] entry causes all processes in the process group to stop or sample when the trap fires. The [pgrp] entry sets the trap in all processes within the process group that contains the code where the trap is set. You can set a default for the action by setting the **Stop All Default** or **Group Trap Default** toggles in the **Traps** menu.

After you enter the trap (by using the **Add** or **Modify** button or by pressing Enter), the full syntax of the specification appears in the field. The **Clear** button clears the **Trap** and **Condition** fields and the cycle fields.

Some typical trap examples are provided in Figure 5-4, page 73. The entries made in the **Trap** field are shown in the left portion of the figure, the trap display in the **Trap Manager** window resulting from these entries is shown on the right, and the trap display shown at the command line in the Main View window is shown at the bottom.

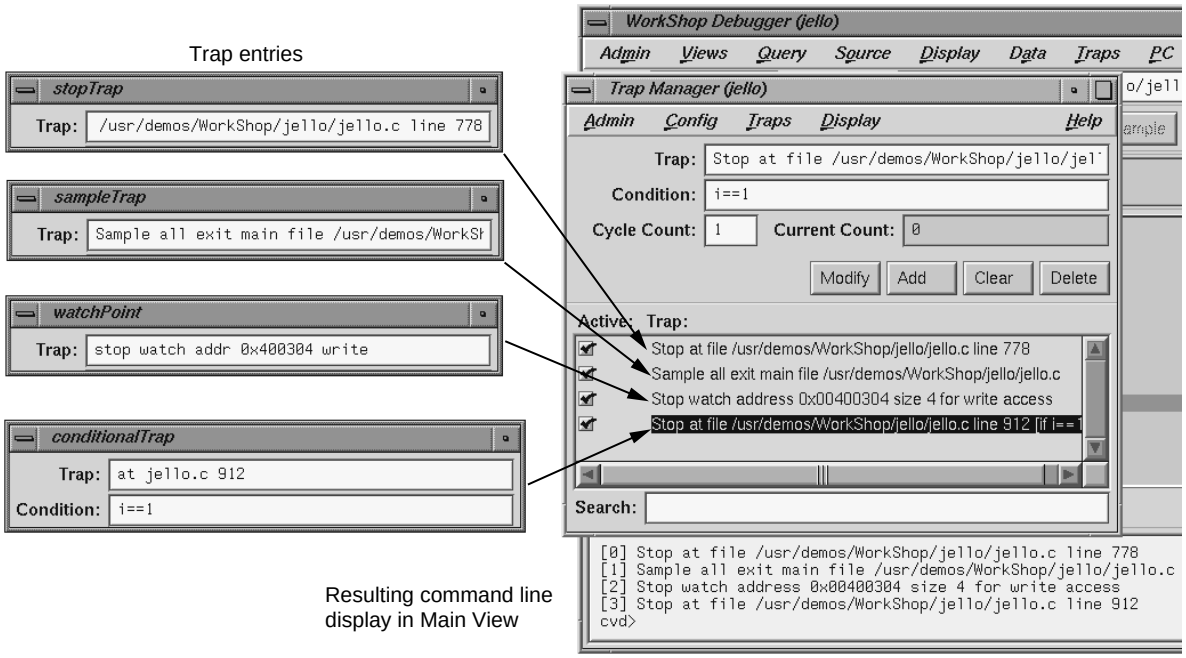

Figure 5-4 Trap Examples

Syntaxes

Specific command syntaxes that may be entered in the **Trap** text field are shown in the following list.

- Setting a Trap in *filename* at *line-number*:

```
[stop|sample] [all][pgrp] at [{file]filename][[line]line-number]
```

  This command sets a trap at the specified line in the specified file, for example: `Trap:  stop at 1449`.

- Setting a Trap on *instruction-address*:

```
[stop | sample] [all] [pgrp] addr instruction-address
```

  This command sets a trap on the specified instruction address. Instruction addresses may be obtained from the **Disassembly View** window, which is brought up from the **Views** submenu on the Main View window menu bar. The addresses may be entered as shown, such as `'0af8cbb8'X`, or as `0x0af8cbb8`. For example:

```
Trap: stop addr 0x0af8cbb8
```

- Setting a Trap on Entry to *function*:

```
[stop|sample] [all] [pgrp] entry function[[file] filename]
```

```
[stop|sample][all] [pgrp] in function[[file]filename]
```

  This command sets a trap on entry to the specified function. For example:

```
Trap: stop entry anneal
```

  or

```
Trap: stop in anneal
```

  If the filename is given, the function is assumed to be in that file's scope.

- Setting a Trap on Exit from *function*:

```
[stop|sample][all][pgrp] exit function[[file]filename]
```

This command sets a trap on exit from the specified function. For example:

```
Trap: stop exit anneal file generic.c
```

If the filename is given, the function is assumed to be in that file's scope.

- Setting a Watchpoint on Specified *expression*:

```
[stop|sample][all][pgrp] watch expression[[for] read|write|execute
  [access]]
```

This command sets a watchpoint on the specified expression (using the address and size of the expression for the watchpoint span). The watchpoint may be specified to fire on `write`, `read`, or `execute` (or some combination thereof). If not specified, the `write` condition is assumed. This syntax has no provision for looking on only a portion of an array. The next syntax item can handle such a request. For example:

```
Trap: stop watch x for write
```

- Setting a Watchpoint for *address* and *size*:

```
[stop|sample][all][pgrp] watch addr[ess]address[[size]size]
  [[for] read|write|execute[access]]
```

This command sets a watchpoint for the specified address and size in bytes. Typically the expression is the name of a variable. The watchpoint may be specified to fire on `write`, `read`, or `execute` (or some combination thereof) of memory in the given span. If not specified, the size defaults to 4 bytes. Also, if not specified, the write condition is assumed. Addresses may be found by choosing the following from the Main View window menu bar: **Views > Variable Browser** or **Views > Structure Browser**.

The window displays addresses for arrays and has options to display addresses for other variables. Addresses may be entered in the form `0x0123fabc` or `'0123fabc'X`. For example:

```
Trap: stop watch addr 0x0123fabc 16
```

If the array you are watching at address `0x0123fabc` is defined such that each element is 4 bytes long, this example trap would be watching 4 adjacent array elements and would cause a stop if any of those 4 elements were updated, since a size of 16 is specified.

- Setting a Trap at *signal-name*:

```
[stop|sample] [all] [pgrp] signal signal-name
```

This command sets a trap upon receipt of the given signal. This is the same as the `dbx`(1) `catch` subcommand. (For a list of signals, or an alternative way to set traps involving signals, see "Setting Traps by Using Signal Panel and System Call Panel", page 79.) For example:

```
Trap: stop signal SGIFPE
```

- Setting a Trap on Entry to *sys-call-name*:

```
[stop|sample] [all] [pgrp] syscall entry sys-call-name
```

This command sets a trap on entry to the specified system call. This is slightly different from setting a trap on entry to the function by the same name. A syscall entry trap sets a trap on entry to the actual system call. A function entry trap sets a trap on entry to the stub function that calls the system call. (For a list of system calls, or an alternative way to set traps involving system calls, see "Setting Traps by Using Signal Panel and System Call Panel", page 79.) For example:

```
Trap: stop syscall entry write
```

- Setting a Trap on Exit from *sys-call-name*:

```
[stop|sample] [all] [pgrp] syscall exit sys-call-name
```

This command sets a trap on exit from the specified system call. This is slightly different from setting a trap on exit from the function by the same name. A syscall exit trap sets a trap on exit from the actual system call. A function exit trap sets a trap on exit from the stub function that calls the system call. (For a list of signals,

or an alternative way to set traps involving signals, see "Setting Traps by Using Signal Panel and System Call Panel", page 79.) For example:

```
Trap: stop syscall exit read
```

- Setting a Trap at *time* Interval:

```
[stop|sample] pollpoint [interval] time [seconds]
```

This command sets a trap at regular intervals of seconds. This is typically used only for sampling. For example:

```
Trap: stop pollpoint 3
```

- Setting a Trap for C++ Exception

```
[stop|sample] exception [all | item] itemname
```

This command sets a trap on all C++ exceptions, or exceptions that throw the base type item.

```
[stop|sample] exception unexpected [all|[item[, item]]]
```

Stops on all C++ exceptions that have either no handler or are caught by an unexpected handler. If you specify *item*, stops on executions that throw the base type *item*.

Setting a Trap Condition

The **Condition** field in the **Trap Manager** window lets you specify the condition necessary for the trap to be fired. A condition can be any legal expression and is considered to be true if it returns a nonzero value when the corresponding trap is encountered.

The expression must be valid in the context in which it will be evaluated. For example, a Fortran condition like **`a .gt .2`** cannot be evaluated if it is tested while the program is stopped in a C function.

There are two possible sequences for entering a trap with a condition:

1. Define the trap.
2. Define the condition.
3. Click **Add**.

or

1. Define the trap.
2. Click **Add**.
3. Define the condition.
4. Click **Modify** (or press `Enter`).

An example of a trap with a condition is shown in Figure 5-4, page 73. The expression `i==1` has been entered in the **Condition** field. (If you were debugging in Fortran, you would use the Fortran expression `i .eq .1` rather than `i==1`.) After the trap has been entered, the condition appears as part of the trap definition in the display area. During execution, any requirements set by the trigger must be satisfied first for the condition to be tested. A condition is true if the expression (valid in the language of the program you are debugging) evaluates to a nonzero value.

Setting a Trap Cycle Count

The **Cycle Count** field in the **Trap Manager** window lets you pass through a trap a specific number of times without firing. If you set a cycle count of *n*, the trap will fire every *n*th time the trap is encountered. The **Current Count** field indicates the number of times the process has passed the trap since either the cycle count was set or the trap last fired. The current count updates only when the process stops.

Setting a Trap with the Traps Menu

The **Traps** menu of the **Trap Manager** window lets you specify traps in conjunction with the Main View or **Source View** windows. Clicking **At Source Line** sets a trap at the line in the source display area that is currently selected. To set a trap at the beginning or end of a function, highlight the function name in the source display and click **Entry Function** or **Exit Function**.

Moving around the Trap Display Area

The trap display area displays all traps set for the current process. There are vertical and horizontal scroll bars for moving around the display area. The **Search** field lets you incrementally search for any string in any trap.

Enabling and Disabling Traps

Each trap has an indicator to its left for toggling back and forth between active and inactive trap states. This feature lets you accumulate traps and turn them on only as needed. Thus, when you do not need the trap, it will not be in your way. When you do need it, you can easily activate it.

Saving and Reusing Trap Sets

The **Load Traps** selection in the **Config** menu lets you bring in previously saved trap sets. This is useful for reestablishing a set of traps between debugging sessions. The **Save Traps...** selection of the **Config** menu lets you save the current traps to a file.

Setting Traps by Using Signal Panel and System Call Panel

You can trap signals by using the **Signal Panel** and set system calls by using **System Call Panel** (see Figure 5-5, page 80).

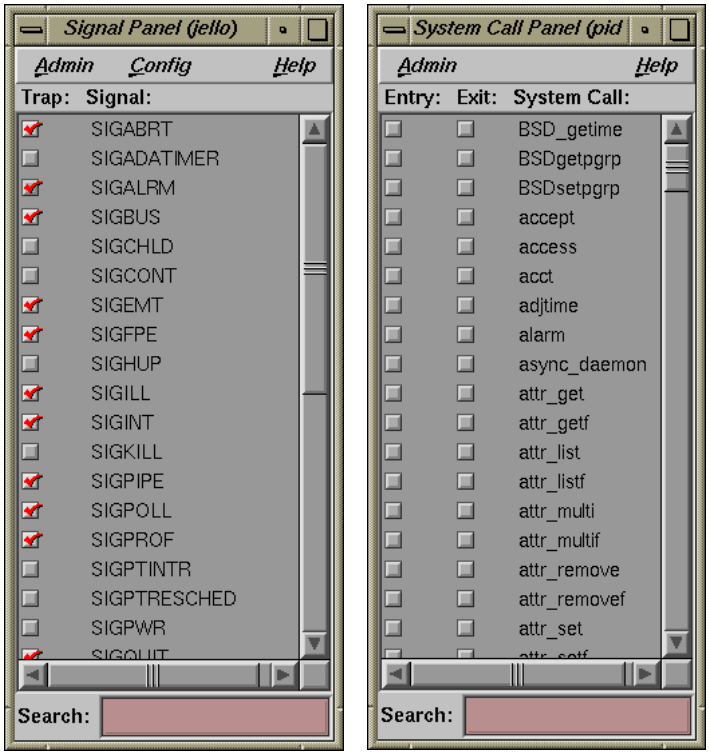

Figure 5-5 Signal Panel and **System Call** Panel

You can select either panel from the **Views** menu of the Main View window menu bar. The **Signal Panel** sets a trap on receipt of the signal(s) selected. The **System Call Panel** sets a trap at the selected entry to or return from the system call.

Note: When debugging IRIX 6.5 pthreads, the **Signal Panel** is inaccessible if more than one thread is active.

Controlling Program Execution

This chapter shows you how to control the execution of your program with WorkShop Debugger. It includes the following topics:

The Main View Window Control Panel

The Main View window control panel allows you to choose an executable, and control its execution:

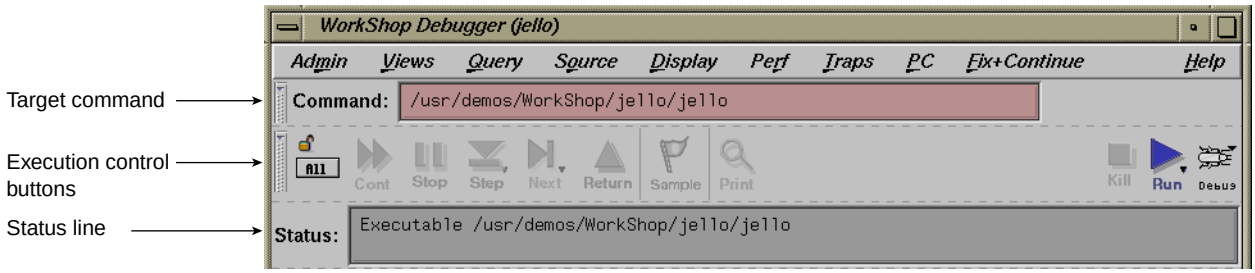

Figure 6-1 The Main View Window Control Panel

Features of the Main View Window Control Panel

The control panel includes the following items:

- **Command** field: use this field to enter shell commands (with arguments) to run your program.
- **Status** field: displays information about the execution status of your program. The top line in this box indicates whether the program is running or stopped. The message **No executable** displays if no executable is loaded. When your program stops at a breakpoint, an additional status line lists the current stack frame.

To see all of the stack frames, select **Views > Call Stack** from the Menu Bar.

Execution Control Buttons

The execution control buttons enable you to control program execution. Most of these buttons are not active until the **Run** button has been selected and the program is executed. The **Print** button does not affect program execution. It is described in Appendix A, "Debugger Reference", page 171.

- **Run**: creates a new process for your program and starts its execution. The **Run** button is also used to re-run a program.
- **Kill**: kills the active process.
- **Stay Focused/Follow Interesting**: if the lock icon is locked, it indicates that the focus of Main View will attempt to stay focused on this thread. If the lock is unlocked, the debugger will follow the interesting thread. This means it will focus on threads that reach a user breakpoint.
- **All/Single**: if set to **All**, the **Cont**, **Stop**, **Step**, **Next**, and **Return** actions apply to all processor or threads. If set to **Single**, then only the currently focused process or thread will be acted upon.
- **Cont**: resumes program execution after a halt and continues until a breakpoint or other event stops execution.
- **Stop**: stops execution of your program. When program execution stops, the current source line is highlighted in the Main View window and annotated with an arrow.
- **Step**: step into function or subroutine calls by default if the function that is stepped into was compiled with `-g` (full debugging information). For libraries like `libc.so`, **step** will not step into it by default.
- **Next**: steps over function or subroutine calls to the next source line. To step a specific number of lines, right-click on this button to display the pop-up menu shown in Figure 6-2, page 83. You can select one of the fixed values or enter your own number of steps by selecting **N**. If you select **N**, the dialog box shown at the right in Figure 6-2, page 83 displays.

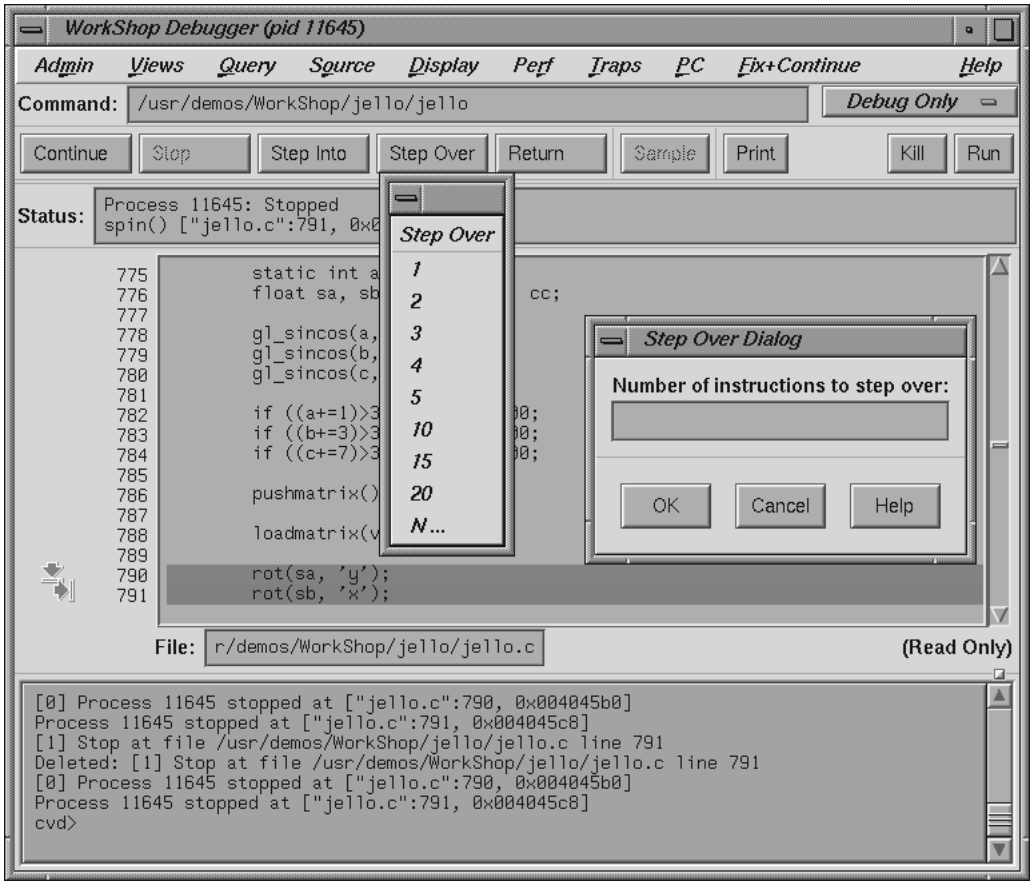

Figure 6-2 Pop-up Menu and **Next Dialog**

- **Return**: executes the remaining instructions in the current function or subroutine, and stops execution at the return from this subprogram.
- **Sample**: collects performance data. Before this button is operative, a performance task must have been previously specified in the **Performance Task** window and data collection must have been enabled.

  For further information about using the Performance Analyzer, see *ProDev Workshop: Performance Analyzer User's Guide*

Controlling Program Execution Using the PC Menu

The **PC** (program counter) menu in the Main View window includes two tools, **Continue To** and **Jump To**, which allow you to control program execution without setting breakpoints.

These tools are inoperative until a process has been executing and is stopped. At that point, you must place your cursor on a source line you wish to target, then select from the **PC** menu depending on your requirements.

- **Continue To**: this tool lets you select a target location in the current program (by placing the cursor in the line). The process proceeds from the current program counter to that point, provided there are no interruptions. It then stops there, as it would for a stop trap. **Continue To** is equivalent to setting a one-time trap. If the process is interrupted before reaching your target location, then the command is cancelled. **Continue To** is useful to move past the end of a `for` or `while` loop that is stepped in, but which has no further interest to you.
- **Jump To**: this tool lets you select a target location in the current program (by placing the cursor in the line). This location must be in the same function. Instead of starting from the current program counter, **Jump To** skips over any intervening code and restarts the process at your target. This is particularly useful if you want to get around bad code or irrelevant portions of the program. It also lets you back up and reexecute a portion of code.

Execution View

The **Execution View** window is a simple shell that lets you set environment variables and inspect error messages. If your program is designed to be interactive using standard I/O, this interaction will take place in the **Execution View** window. Any standard I/O that is not redirected by your Target Command is displayed in the **Execution View** window.

When you launch the debugger, the **Execution View** window is launched in iconified form.

Viewing Program Data

After you set traps (breakpoints) in your program, use the **Run** button to execute your program. When a trap stops a process, you can view your program data using the tools described in this chapter. This chapter covers:

The Debugger also lets you examine data at the machine level. The tools for viewing disassembled code, machine registers, and data by specific memory location are described in Appendix A, "Debugger Reference", page 171.

Traceback Through the Call Stack Window

The **Views** menu may be used to bring up the **Call Stack** window. This window displays the functions/subroutines (that is, the “frames”) in the call stack when the process associated with your program has stopped. This display provides a traceback of subprograms from the system routine which starts your executable, found at the bottom of the list, to the routine in which you are currently stopped, found at the top of the list.

If the trap/breakpoint at which you are currently stopped is located in your source code, that code is displayed in the source pane of the Main View window. If the trap/breakpoint is located in a system routine, the **Call Stack** allows you to double-click on another routine’s name to bring up that routine’s source and associated data. A typical **Call Stack** window is shown in Figure 7-1, page 86.

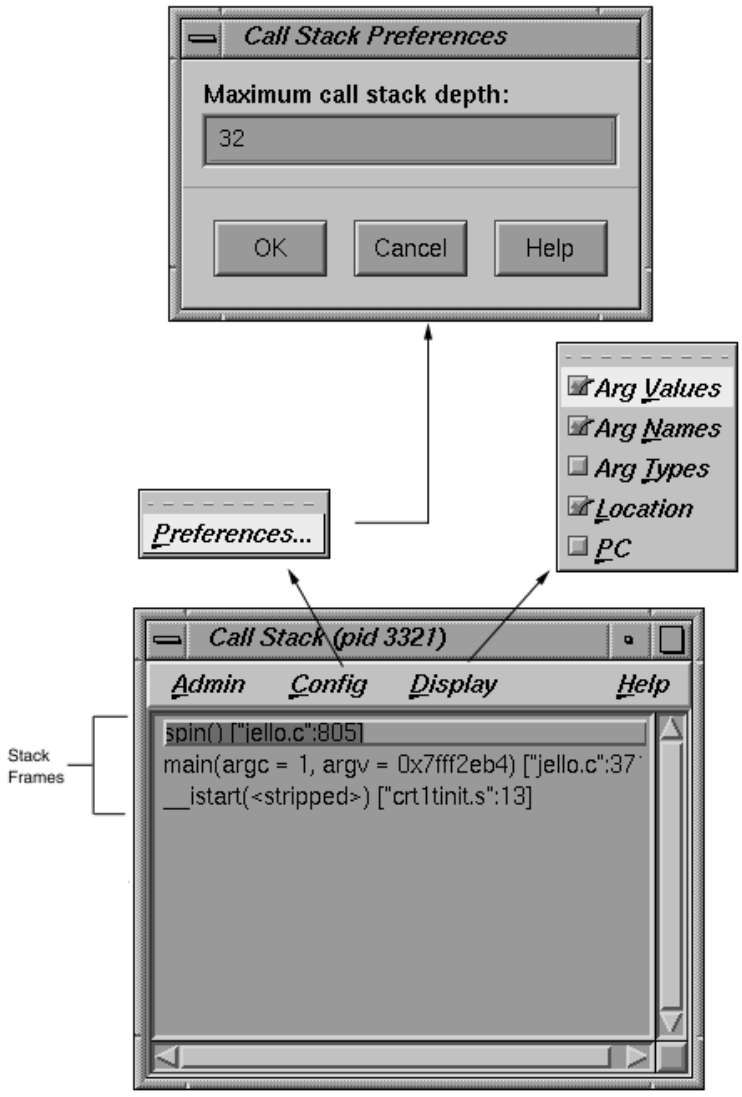

Figure 7-1 Call Stack Window

The **Call Stack** window lets you see the argument names, values, types, and locations of functions, as well as the program counter (PC).

If symbolic information for the arguments has been stripped from the executable file, the label `<stripped>` appears in place of the arguments. By default call stack depth is set to 10, but you can reset the depth of the **Call Stack** by selecting **Config > Preferences** from the Main View window.

To move through the call stack, double-click a frame in the stack. The frame becomes highlighted to indicate the current context. The source display in the Main View or **Source View** windows scrolls automatically to the location where the function was called and any other active views update.

The source display has two special annotations:

- The location of the current program state is indicated by a large arrow. This represents the PC (program counter).
- The location of the call to the function selected in the **Call Stack** window is indicated by a smaller arrow. This represents the current context, and the source line is highlighted.

Figure 7-2, page 88, illustrates the correspondence between a frame and the source code when a frame is clicked in the **Call Stack** window. In this example, the stack frame `spin` has been selected; the Main View display scrolls to the place where the trap occurred. If the second stack (`main`) had been selected, the window would have scrolled to the place where the function `main` calls `spin`.

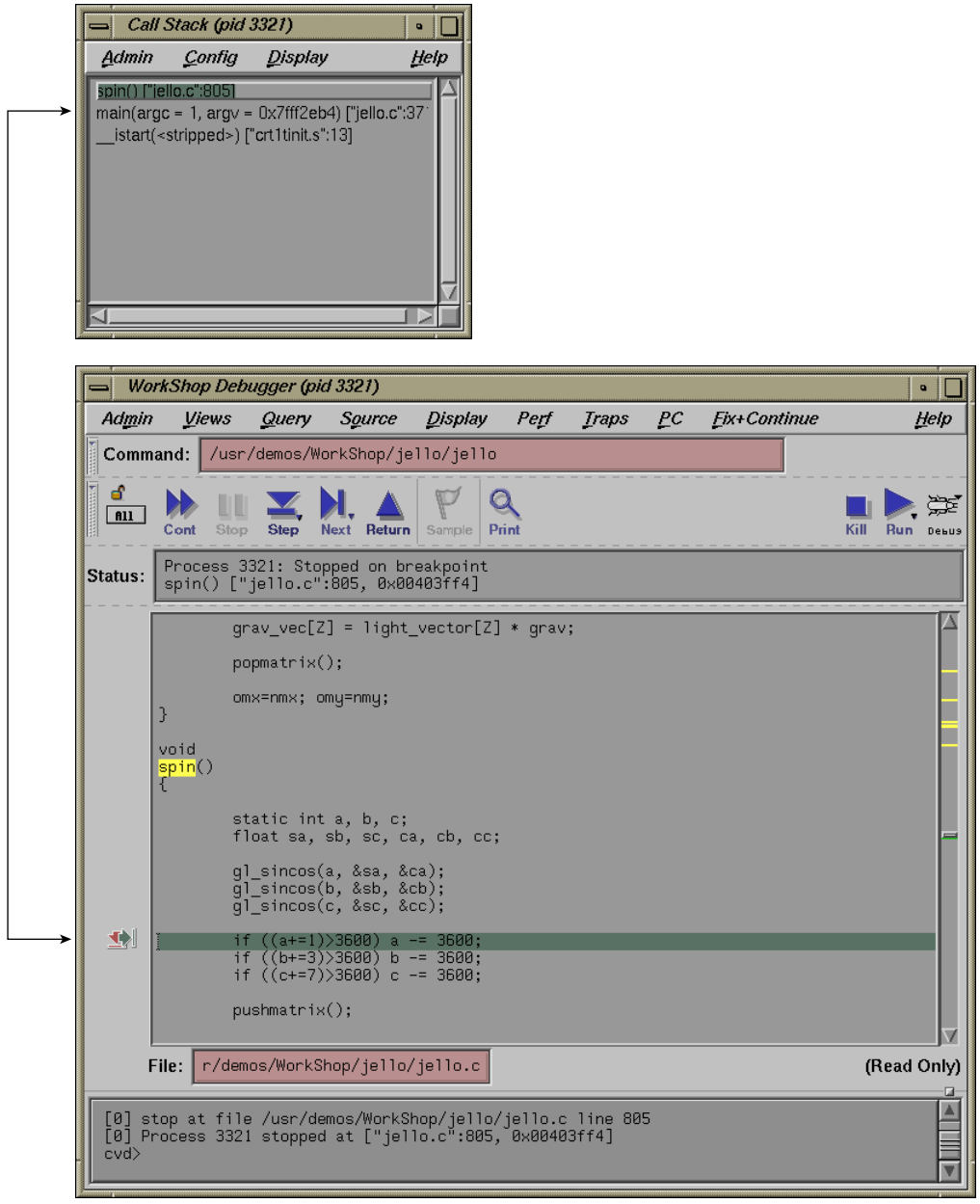

Figure 7-2 Tracing through **Call Stack**

Options for Viewing Variables

The WorkShop Debugger provides several options for viewing variables or expressions involving these variables. In this section only brief descriptions of these options will be provided along with references where further information may be obtained elsewhere in this document.

Using the cvd Command Line

At the bottom of the Main View window is the `cvd` command/message pane. Here, you can enter `print xxx` commands to obtain the current value of variable `xxx`.

For examples of these commands, see "Viewing Variables Using the `cvd` Command Line", page 18.

For the syntax of these commands (such as `assign`, `print`, `printd`, `printo`, `printx`, and so on), see "Syntax for dbx-style Commands", page 313.

Using Click to Evaluate

In the Main View window's source pane, a click of the right mouse button will bring up a pop-up menu from which you can select **Click to Evaluate**. When this option is on, a click on a variable will cause its value to appear. If you click on a subscript variable, its address will appear. If you hold down the left mouse button and wipe across a variable and its subscripts to highlight them, the current value will be displayed. The same is true if an expression in the source is highlighted.

Using the Array Browser

To view values of arrays, select **Views > Array Browser** from the Main View menu bar.

This calls up the **Array Browser** window. When the name of an array is entered into the **Array** field, the values of 1–dimensional arrays or the values in a selected plane of a multi-dimensional array are displayed.

See "Viewing Variables Using the **Array Browser**", page 22 for short description of the Array Browser. Also, "Array Browser Window", page 255, "**Array Browser** Window", provides a more detailed description and graphic of the window and associated submenus.

Using the Structure Browser

To view C structures, select **Views > Structure Browser** from the Main View menu bar.

This calls up the **Structure Browser** window. When the name of a structure is entered into the **Expression** field, the objects in the structure display in the lower portion of the window. "Structure Browser Window", page 277, “**Structure Browser** Window”, provides a more detailed description and graphic of the window and associated submenus.

Using the Variable Browser

To call up the **Variable Browser** window, select **Views > Variable Browser** from the Main View menu bar.

This window lists the names and values of variables associated with the current routine in which the process has stopped.

See "Viewing Variables Using the **Variable Browser**", page 20 for a short description of the Variable Browser. Also, "Variable Browser Window", page 287, “**Variable Browser** Window”, provides a more detailed description and graphic of the window and associated submenus.

Using the Expression View

Select the following from the Main View menu bar to bring up the **Expression View** window: **Views > Expression View**.

Expressions may be entered into this window in the **Expression** column; and the corresponding **Results** entry will display the value of the evaluated expression using values of variables associated with the current routine in which the process has stopped.

See "Viewing Variables Using the **Expression View** Window", page 20 for a short description of the Expressions View window. Also, "Expression View Window", page 273, **Expression View** Window, provides a more detailed description and graphic of the window and associated submenus.

The next section of this chapter, “Evaluating Expressions”, gives further details on how to create acceptable expressions for the **Expression View**.

Evaluating Expressions

You can evaluate any valid expression at a stopping point and trace it through the process. Expressions are evaluated by default in the frame and language of the current context. Expressions may contain data names or constants; however, they may not contain names known only to the C preprocessor, such as in a `#define` directive or a macro.

To evaluate expressions, you can use **Expression View**, which lets you evaluate multiple expressions simultaneously, updating their values each time the process stops.

You can also evaluate expressions from the command line. See "Debugger Command Line", page 313 for more information.

Expression View Window

The **Expression View** window has two pop-up menus, the **Language** menu and the **Format** menu:

- The **Language** menu is invoked by holding down the right mouse button while the cursor is in the **Expression** column.
- The **Format** menu is displayed by holding down the right mouse button in the **Result** column.

To specify the expression to be evaluated, click in the **Expression** column and then enter the expression in the selected field. It must be a valid expression in the current or selected language: Ada, C, C++, or Fortran. To change languages, display the **Language** menu and make your selection. When you press `Enter`, the result of the expression is displayed in the **Result** column.

To change the type of result information displayed in the right column, hold down the right mouse button over the right column. This displays the **Format** menu. From here you can select the following:

- Select the **Default Value** menu to see the value as decimal, unsigned, octal, hex, float, char, or string characters.
- Select the **Type Address Of** menu to display the address in decimal, octal, or hexadecimal.
- Select **Bit Size** to specify the size of the result, in bits.

Caution: The Debugger uses the symbol table of the target program to determine variable type. Some variables in libraries, such as `errno` and `_environ`, are not fully described in the symbol table. As a result, the Debugger may not know their types. When the Debugger evaluates such a variable, it assumes that the variable is a fullword integer. This gives the correct value for fullword integers or pointers, but the wrong value for non-fullword integers and for floating-point values.

To see the value of a variable of unknown type, use C type cast syntax to cast the address of the variable to a pointer that points to the correct type. For example, the global variable `_environ` should be of type `char**`. You can see its value by evaluating `*(char***)&_environ`.

After you display the current value of the expression, you may find it useful to leave the window open so that you can trace the expression as it changes value from trap to trap (or when you change the current context by double-clicking in the call stack). Like other views involved with variables, **Expression View** has variable change indicators for value fields that let you see previous values, as shown in the following figure:

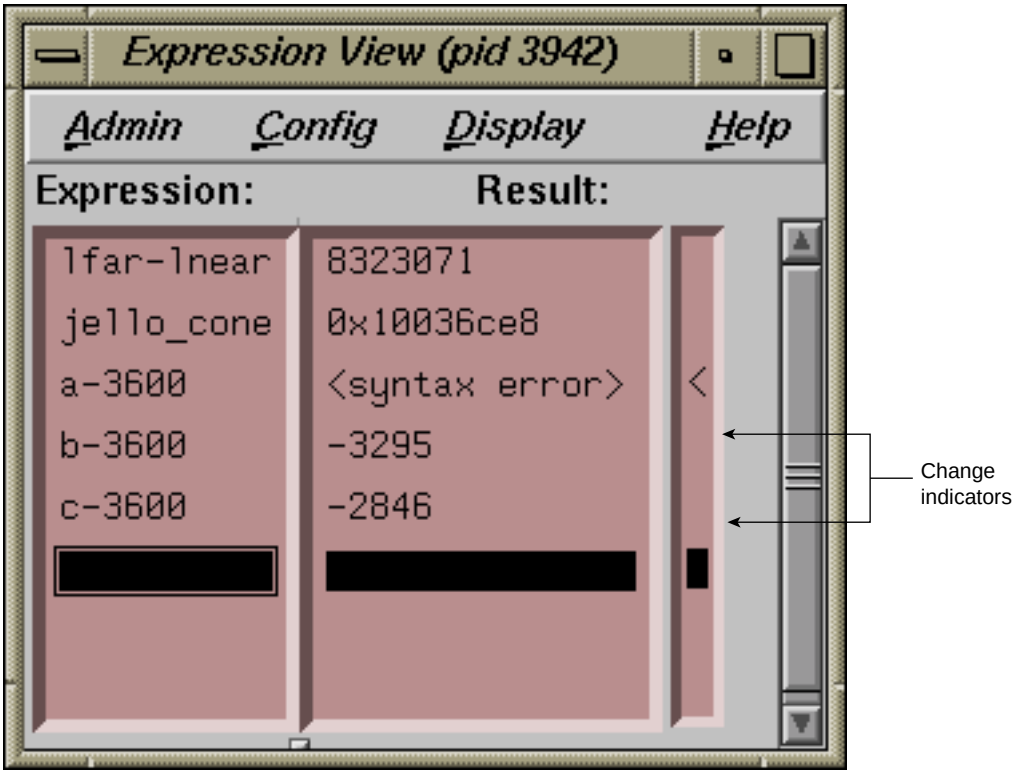

Figure 7-3 Change Indicator in `Expression View`

Another useful technique is to save your expressions to a file for later reuse. To save expressions, select **Config > Save Expressions** from the Main View menu bar.

To load expressions, select **Config > Load Expressions** from the Main View menu bar.

Assigning Values to Variables

To assign a value to a variable, click the left column of the **Expression View** window and enter the variable name. The current value appears in the right column. If this **Result** field is editable (highlighted), you can click it and enter a new value or legal expression. Press `Enter` to assign the new value. You can perform an assignment to any expression that evaluates to a legal `lvalue` (in C). The C operator "=" is not

valid in **Expression View**. Valid expression operations are shown in the following paragraphs.

Evaluating Expressions in C

The valid C expressions are shown in Table 7-1.

Table 7-1 Valid C Operations

Operation	Symbol
Arithmetic[1]	`+ - ++ --`
Arithmetic (binary)	`+ - * / %`
Logical	`&& || !`
Relational	`< > <= >= == !=`
Bit	`& | ^ << >> ~`
Dereference	`*`
Address	`&`
Array indexing	`[ ]`
Conditional	`? :`
Member extraction[2]	`. ->`
Assignment[3]	`= += -= /= %= >>= <<= &= ^= |=`

1 Unary - increment and decrement do not have side-effects

2 These operations are interchangeable.

3 A new assignment is made at each stepping point. Use Assignments with caution to avoid inadvertently modifying variables.

Operation	Symbol
Sizeof	
Type-cast	
Function call	

C Function Calls

Function calls can be evaluated in expressions, as long as enough actual parameters are supplied. Arguments are passed by value. Following the rules of C, each actual parameter is converted to a value of the same type as the formal parameter, before the call. If the types of the formal parameters are unknown, integral arguments are widened to full words, and floating-point arguments are converted to doubles.

Functions may return pointers, scalar values, unions, or structs. Note that if the function returns a pointer into its stack frame (rarely a good programming practice), the value pointed to will be meaningless, since the temporary stack frame is destroyed immediately after the call is completed.

Function calls may be nested. For example, if your program contains a successor function `succ`, the Debugger will evaluate the expression `succ(succ(succ(3)))` to 6.

Evaluating Expressions in C++

C++ expressions may contain any of the C operations. You can use the word `this` to explicitly reference data members of an object in a member function. When stopped in a member function, the scope for `this` is searched automatically for data members. Names may be used in either mangled or de-mangled form. Names qualified by class name are supported (for example, `Symbol::a`).

If you wish to look at a static member variable for a C++ class, you need not specify the variable with the class qualifier if you are within the context of the class. For example, you would specify `myclass::myvariable` for the static variable *myvariable* outside of class *myclass* and `myvariable` inside *myclass*.

Limitations

Constructors may be called from **Expression View**, just like other member functions. To call a constructor, you must pass in a first argument that points to the object to be created. C++ function calls have the same possibility of side effects as C functions.

Evaluating Expressions in Fortran

You can enter any Fortran expression under the **Expression** heading and its current value will appear in the same row under the **Results** column. Fortran expressions may contain any of the arithmetic, relational, or logical operators. Relational and logical operator keywords may be spelled in upper case, lower case, or mixed case.

The usual forms of Fortran constants, including complex constants, may be used in expressions. String constants and string operations, however, are not supported. The operators in Table 7-2 are supported on data of integer, real, and complex types.

Table 7-2 Valid Fortran Operations

Operation	Symbol
Arithmetic (unary)	`- +`
Arithmetic (binary)	`- + * / **`
Logical	`.NOT. .AND. .OR. .XOR. .EQV .NEQV.`
Relational	`.GT. .GE. .LT. .LE. .EQ. .NE.`
Array indexing	`( )`
Intrinsic function calls (except string intrinsics)	
Function subroutine calls	
Assignment[4]	`=`

[4] A new assignment is made at each stepping point. Use assignments with caution to avoid inadvertently modifying variables.

Fortran Variables

Names of Fortran variables, functions, parameters, arrays, pointers, and arguments are all supported in expressions, as are names in common blocks and equivalence statements. Names may be spelled in upper case, lower case, or mixed case.

Fortran Function Calls

The Debugger evaluates function calls the same way that compiled code does. If it can be, an argument is passed by reference; otherwise, a temporary expression is allocated and passed by reference. Following the rules of Fortran, actual arguments are not converted to match the types of formal arguments. Side effects can be caused by Fortran function calls. A useful technique to protect the value of a parameter from being modified by a function subroutine is to pass an expression such as `(parameter + 0)`instead of just the parameter name. This causes a reference to a temporary expression to be passed to the function rather than a reference to the parameter itself. The value is the same.

Debugging with Fix+Continue

Fix+Continue allows you to make changes to a C++ program you are debugging without having to recompile and link the entire program. With Fix+Continue you can edit a function, parse the new function, and continue execution of the program being debugged.

Fix+Continue is an integral part of the Debugger. You issue Fix+Continue commands graphically from the **Fix+Continue** submenu of the Main View window, or from the `cvd` command line prompt in the Command/Message pane of the Main View window.

This chapter provides an introduction to the Fix+Continue functionality as well as a tutorial to demonstrate many of the Fix+Continue functions.

Fix+Continue Functionality

Fix+Continue lets you perform the following activities:

- Redefine existing function definitions.
- Disable, re-enable, save, and delete redefinitions.
- Set breakpoints in redefined code.
- Single-step within redefined code.
- View the status of changes.
- Examine differences between original and redefined functions.

A typical Fix+Continue cycle proceeds as follows:

1. You redefine a function with Fix+Continue. When you continue executing the program, the Debugger attempts to call the redefined function. If it cannot, an information pop-up window appears and the redefined function will be executed the next time the program calls that function.
2. You redefine other functions, alternating between debugging, disabling, re-enabling, and deleting redefinitions. You might save function redefinitions to their own files, or save files to a different name, to be used later with the present or with other programs.

During debugging you can review the status of changes by listing them, showing specific changes, or looking at the Fix+Continue **Status View**. You can compare changes to an individual function or to an entire file with the compiled versions. When you are satisfied with the behavior of your application, save the changed file as a replacement for the compiled source.

Fix+Continue Integration with Debugger Views

Fix+Continue interacts with the following **Views**:

- The **Views** main view, the **Source View**, and **Fix+Continue Status** windows distinguish between compiled and redefined code, and allow editing in redefined code.
- The following status windows elements work with redefined code:
  - **Call Stack** window
  - **Trap Manager**
  - Debugger command line

How Redefined Code Is Distinguished from Compiled Code

Redefined functions have an identification number and special line numbers. They are color-coded according to their state (that is, edited, parsed, and so on).

Line numbers in the *compiled* file stay the same, no matter how redefined functions change. However, when you begin editing a function, the line numbers of the function body are represented in decimal notation (*n*.1, *n*.2, ..., *n*.*m*), where *n* is the compiled line number where the function body begins, and *m* is the line number relative to the beginning of the function body, starting with the number `1`.

The **Call Stack** window and the Trap Manager functions both use function-relative decimal notation when referring to a line number within the body of a redefined function.

The Debugger command line reports ongoing status. In addition to providing the same commands available from the menu, edit commands allow you to add, replace, or delete lines from files. Therefore, you can operate on several files at once.

The Fix+Continue Interface

You can access Fix+Continue through the **Fix+Continue** menu. It includes three supporting windows: **Status, Message**, and **Build Environment**. These windows are part of Fix+Continue, and do not operate unless it is installed.

Debugger with Fix+Continue Support

Without Fix+Continue, the Debugger source views are `Read-Only` by default. That is so you can examine your files with no risk of changing them. When you select **Edit** from the **Fix+Continue** menu, the Debugger source code status indicator (in the lower-right corner of the Debugger window) remains `Read-Only`. This is because edits made using Fix+Continue are saved in an intermediate state. Instead, you must choose **Save File > Fixes As** to save your edits.

When you edit a function, it is highlighted in color; and if you switch to the compiled version of your code, the color changes to show that the function has been redefined. If you try to edit the compiled version of your code, the Debugger beeps indicating `Read-Only` status.

When you have completed your edits and want to see the results, select **Parse and Load**. When the parse and load has executed successfully, the color changes again. If the color does not change, there may be errors: check the **Message Window**.

Change ID, Build Path, and Other Concepts

The Fix+Continue features finding files and accessing functions through ID numbers as follows:

- Each redefined function is numbered with a change ID. Its status may be shown as **redefined**, **enabled**, **disabled**, **deleted**, or **detached**.
- Fix+Continue needs to know the location of `include` files and other parameters specified by compiler build flags. You can set the build environment for all files or for a specific file. You can display the current build environment from the **Fix+Continue** menu, the command line, or the **Fix+Continue Status** Window. When you finish a Fix+Continue session, you can unset the build environment.
- Output from a successful run is displayed in the **Execution View**. This functionality is the same as it is in the Debugger without Fix+Continue.

Restrictions on Fix+Continue

Fix+Continue has the following restrictions:

- When you work with C code, you must use the `-o32` compiler option.
- Fix+Continue does not support C++ templates.
- You may not add, delete, or reorder local variables in a function.
- You may not change the type of a local variable.
- You may not change a local variable to be a register variable and vice- versa.
- You may not add any function calls that increase the size of the parameter area.
- You may not add an `alloca` function to a frame that did not previously use an `alloca` function.
- Both the old and new functions must be compiled with the `-g` option.

  In other words, the layout of the stack frames of both the old and new functions must be identical for you to continue execution in the function that is being modified. If not, execution of the old function continues and the new function is executed the next time the function is called.
- If you redefine functions that are in but not on top of the call stack, the modified code will not be executed when they combine. Modified functions will be executed only on their next call or on a rerun.

  For example, consider the following call stack:

  ```
  foo()
  bar()
  foobar()
  main()
  ```

  – If you redefine `foo()`, you can continue execution, provided that the layout of the stack frames are the same.

  – If you redefine `main()` after you have begun execution, the redefined main() will be executed only when you rerun.

  – If you redefine `bar()` or `foobar()`, the new code will not be executed when `foo()` returns. It will be executed only on the next call of `bar()` or `foobar()`.

Fix+Continue Tutorial

This tutorial illustrates several features of Fix+Continue. The demo files included in `/usr/demos/WorkShop/time1` contain the complete C++ source code for the program `time1`. Use this program for your tutorial. Here you will see how Fix+Continue can modify functions without recompiling and linking the entire program.

This section contains the following subsections:

Setting up the Sample Session

For this tutorial, use the demo files in the `/usr/demos/WorkShop/time1` directory that contains the complete source code for the C++ application `time1`. To prepare for the session, you must create the fileset and launch Fix+Continue from the Debugger as shown below:

1. Enter the following commands:

   ```
   % mkdir demos/time1
   % cd demos/time1
   % cp /usr/demos/WorkShop/time1/* .
   % make time1
   % cvd time1 &
   ```

   The `cvd` command brings up the Debugger, from which you can use the Fix+Continue utility. The **Execution View** icon and the Main View window appear. Note that the Debugger shows a source code status indicator of `(Read Only)`.

2. Open the **Execution View** window and position it next to the Main View window.

3. Click **Run** to run `time1`.

   The **Execution View** shows the program output (see Figure 8-1).

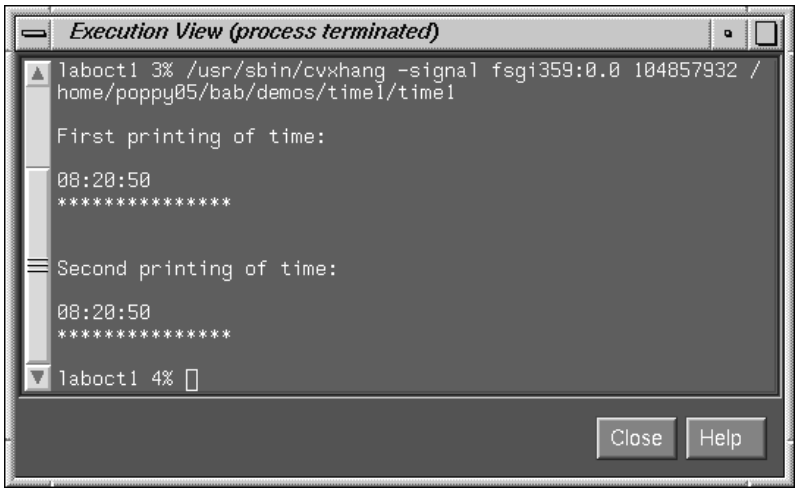

Figure 8-1 Program Results in **Execution View**

Redefining a Function: `time1` Program

In this section, you will do the following in the `time1` program:

- Edit a C/C++ function.
- Change the code of an existing C/C++ function and then parse and load the function, rebuilding your program to see the effect of your changes on program output (without recompiling).
- Save the changed function to its own separate file.

Editing a Function

Perform the following in the Debugger Main View window (the `time1` program should be displayed) to edit a function:

1. Select **Display > Show Line Numbers** from the Main View window menu bar to show line numbers.
2. Click on the source annotation column to the left of line 17 to set a breakpoint at that point.

3. Set a breakpoint at line 20 from the `cvd` command line as follows:

   ```
   cvd> stop at 20
   ```

4. Click on the **Run** button to execute the program.

   The following output appears in the **Execution View** window:

   ```
   First printing of time:

   08:20:50
   ***************

   Second printing of time:
   ```

5. Enter the following at the `cvd` command line to choose a class member function to edit (in this case, printTime, a C++ member function of class Time):

   ```
   cvd> func Time::printTime
   ```

   This command opens the `time1/time1.c` file, which contains the implementation of class Time. The cursor is placed at the beginning of the `printTime` function. SSee Figure 8-2, page 106. The syntax of the `func` command is as follows:

   - For C++ class member functions:

     `cvd> func` *`className`*`::`*`classMemberFunction`*

   - For all other C/C++ functions:

     `cvd> func` *`functionName`*

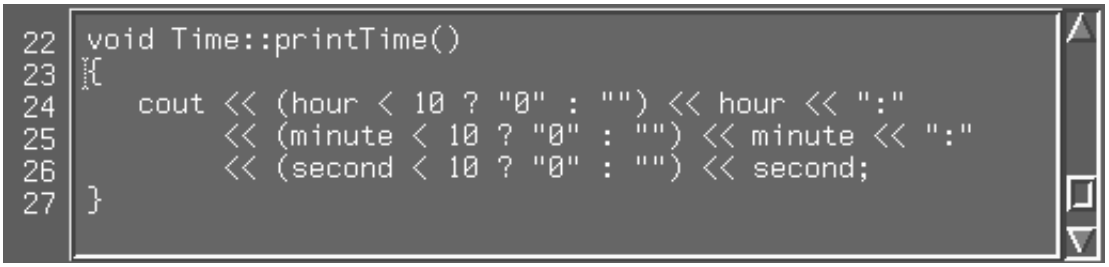

Figure 8-2 Selecting a Function for Redefinition

6. Select **Fix+Continue > Edit** from the menu bar to highlight the function to be edited. You can also use `Alt-Ctrl-e` to do this.

   Note the results as shown in Figure 8-3, page 106. Line numbers changed to a decimal notation based on the first line number of the function body. The function body highlights to show that it is being edited. The line numbers of the rest of the file are not affected.

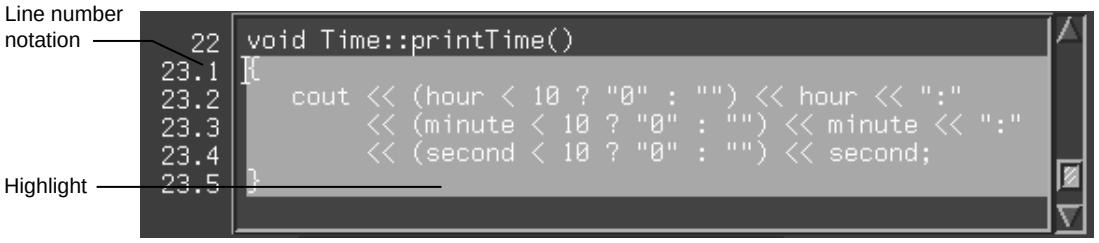

Figure 8-3 Redefined Function

Changing Code

1. To change the time output as shown in Step 5 in "Editing a Function", page 104, delete the `0` from `"0"` in line 23.2.
2. Select **Fix+Continue > Parse And Load** from the menu bar to parse the modified function and load it for execution.

An icon for the **Fix+Continue Error Messages** window displays and the following message appears in the **cvd** window:

```
Change id: 1 modified
```

If there are errors:

a. Go to the error location(s) by double-clicking the related message line in the **Fix+Continue Error Messages** window.

b. Correct the errors.

c. Repeat steps 1 and 2.

Continue to step 3 when you see the change ID and the following messages:

```
Change id: 1 redefined
Change id: 1 saved func
Change id: 1 file not saved
Change id: 1 modified
```

The new function value is not active until the function is called.

3. Click on the **Continue** button to continue program execution.

   The following output appears in the **Execution View** window:

```
Second printing of time:

8:20:50
***************
```

   Notice how the time printout has changed from `08:20:50` to `8:20:50`.

Deleting Changed Code

To cancel any of your changes, you must bring up the source file in which the change was made and perform the following steps:

1. Enter the following at the `cvd` command line:

```
cvd> func Time::printTime
```

2. Select **Fix+Continue > Delete Edits** from the menu bar to delete your changes.

   The **Verify before deleting** dialog displays.

3. Click **OK** in the **Verify before deleting** dialog.

   Your deletion is complete.

Changing Code from the Debugger Command Line

You can redefine a C++ class member function from the Debugger command line as follows:

1. Click on the **Kill** button in the Main View window.
2. Click on the **Run** button to re-run the program.
3. Choose a class member function (in this case, printTime) to edit by entering the following at the **cvd** command line:

   ```
   cvd> func Time::printTime
   ```

4. Use the `redefine` command to edit the function:

   ```
   cvd> redefine Time::printTime
   ```

   Note the results as shown in Figure 8-3, page 106. Here, line numbers have changed to decimal notation based on the first line number of the function body. Note also that the command line prompt has changed.

5. Enter the following at the prompt:

   "*/path/name/*`time1.C":23.1> .`

6. Change the function source by entering the following at the command line:

   ```
   cvd> replace_source "time1.C":23.2
   "time1.C:23.2> cout << (hour < 10 ? "" : "") << hour << ":"
   "time1.C:23.3> .
   ```

   Parse and Load is executed at this point. You exit out of the function edit mode are return to the main source code. The following messages appear in the command/message pane:

   ```
   Change id: 2 redefined
   Change id: 2 save func
   Change id: 2 file not saved
   Change id: 2 modified
   Change id: 2 , build results:
          2     enabled .../time1.C Time::printTime(void)
   ```

Note: 2 in these messages is the redefined function ID. You will use this ID in the procedure in "Switching between Compiled and Redefined Code", page 112. The new function value is not active until the function is called.

7. Continue execution by entering the following command at the `cvd` command line:

   ```
   cvd> continue
   ```

   The following displays in the **Execution View** window:

   ```
   Second printing of time:

   8:20:50
   ***************
   ```

If you prefer to use the command line, experiment with `add_source` and other commands that give you the same functionality described for the menu commands. For details on each command, see "Debugger Command Line", page 313.

Saving Changes

Your original source files are not updated until the changed source file is saved. You could save redefined function changes to the `time1.C` file. However, if you did, the file would not match the tutorial. So perform the following steps:

1. Enter the following command:

   ```
   cvd> func Time::printTime
   ```

2. Select **Fix+Continue > Save As** from the menu bar.

   A *file_name* dialog box opens.

3. The dialog box enables you to save your file changes back to the original source files or save them to a different file. However, since you do not want to save your changes, press the **Cancel** button on the bottom of the dialog box.

Note: You should wait until you are finished with Fix+Continue before you save your changes. In addition to the method described above, you can also save your changes by selecting **Fix+Continue > Save All Files**.

Setting Breakpoints in Redefined Code

To see how the Debugger works with traps in redefined code, this section shows you how to set breakpoints, run the Debugger, and view the results (Figure 8-4, page 111).

1. Reset to the beginning of program execution by entering the following at the `cvd` command line:

   ```
   cvd> kill
   cvd> run
   ```

2. Bring up the `time1.C` source file by entering the following at the `cvd` command line:

   ```
   cvd> func Time::printTime
   ```

3. Select **Fix+Continue > Edit** from the menu bar. You can also use the `Alt-Ctrl-e` accelerator to do this.

4. Enter the following after line **23.4** in the source pane of the Main View window:

   ```
   cout << " AM"<< endl;
   ```

5. Select **Fix+Continue > Parse And Load** from the menu bar.

6. Bring up the `time1.C` source file again by entering the following command at the `cvd` command line:

   ```
   cvd> func Time::printTime
   ```

7. Set a breakpoint at line **23.6** by entering the following message at the `cvd` command line:

   ```
   cvd> stop at 23.6
   ```

   The following message appears in the Command/Message pane:

   [2] Stop at file */path/name*/time1.C line 23.6

8. Click on the **Cont** button in the Main View window. The following figure shows how the breakpoint has been reached in the redefined code:

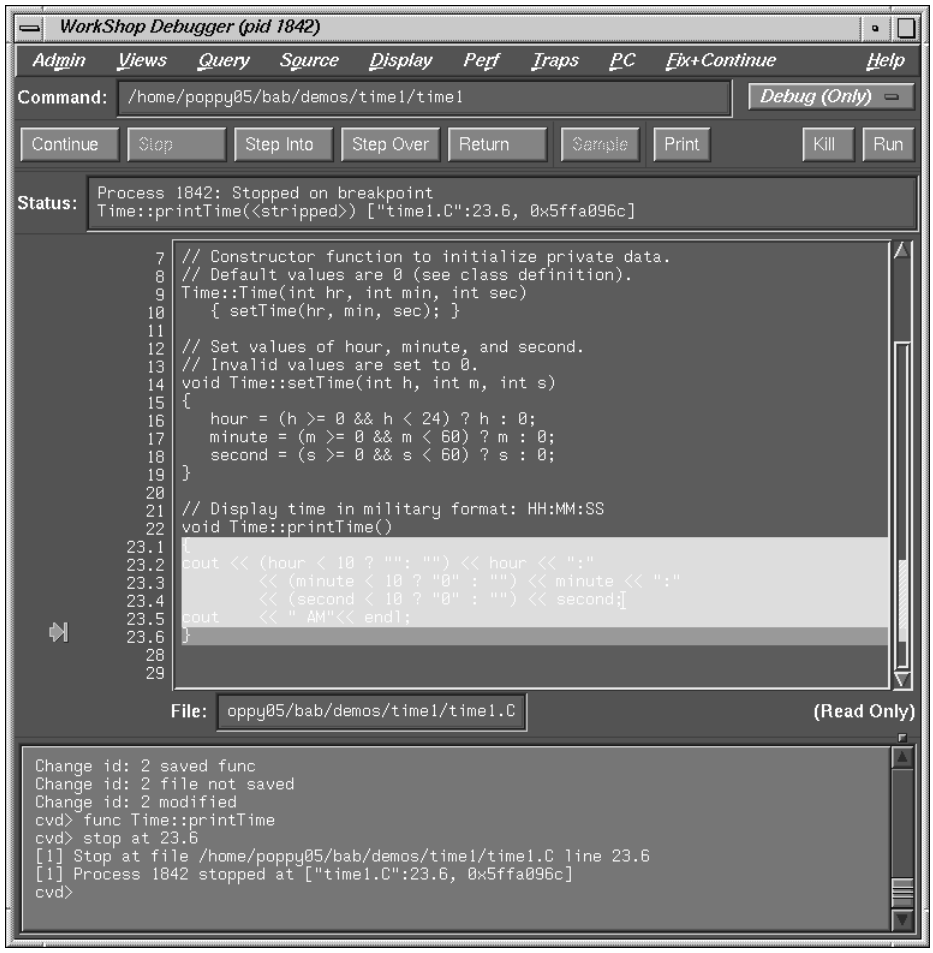

Figure 8-4 Stopping after Breakpoints in Redefined Code

9. Select the following from the menu bar to see the results of continuing to the breakpoint: **Views > Call Stack**.
10. Select the following from the menu bar to view the locations of the breakpoints: **Views > Trap Manager**.
11. Remove the breakpoint by clicking on the source annotation column to the right of line **23.6**.

To view status, select the following from the menu bar: **Fix+Continue > View > Status Window**. The **Fix+Continue Status** window opens.

Comparing Original and Redefined Code

You can use Fix+Continue to compare original code with and modified code. This section shows you several ways to view your changes.

Switching between Compiled and Redefined Code

Follow these steps to see how the redefined code affects your executable:

1. Click the **Run** button to view your redefined code. If you are following procedures in order in this section, you should see the following display:

   ```
   8:20:50 AM
   ```

2. Enter the following at the `cvd` command line:

   ```
   cvd> func Time::printTime
   ```

3. Select **Fix+Continue > Parse and Load** from the menu bar.

4. Select **Fix+Continue > Edited<–>Compiled** from the menu bar to disable your changes.

5. Click the **Continue** button to see the printing of `Time` as in the original executable. The following displays:

   ```
   08:20:50
   ```

6. Re-enter the following at the **cvd** command line:

   ```
   cvd> func Time::printTime
   ```

7. Select **Fix+Continue > Edited<–>Compiled** from the menu bar to re-enable your changes.

8. Click on the **Run** button to see the changed printout of `Time`. The following displays:

   ```
   08:20:50 AM
   ```

Comparing Function Definitions

1. Place the cursor in the `time1.C` function.
2. Select **Fix+Continue > Show Difference > For Function** from the menu bar.

   A window opens to display an `xdiff` comparison of the files as follows:

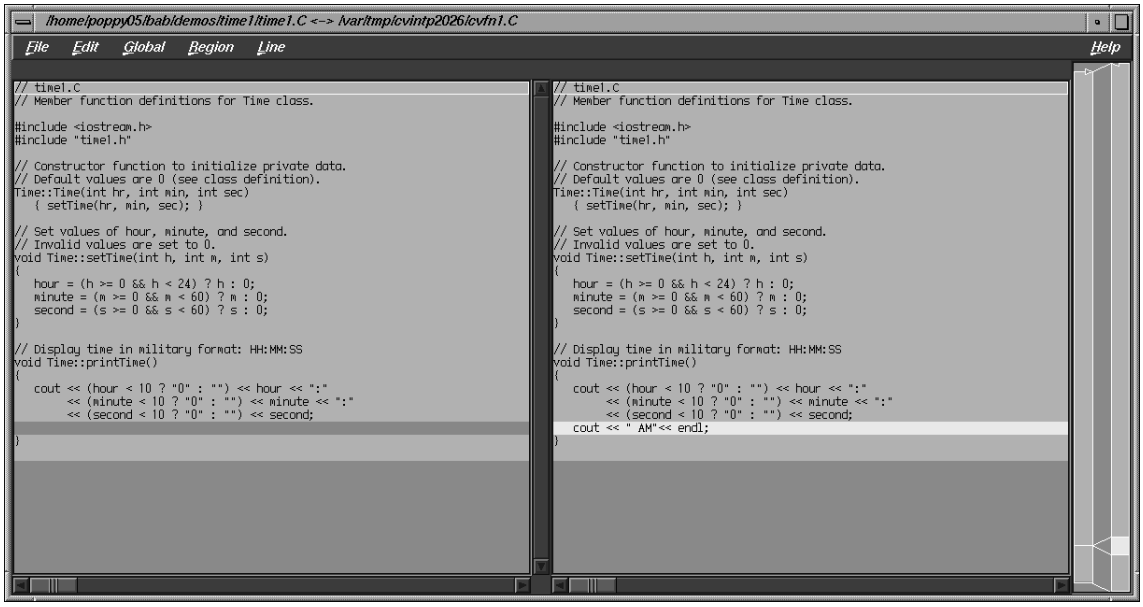

Figure 8-5 Comparing Compiled and Redefined Function Code

You can get the same result by entering the **`show_diff`** # command from the Debugger command line, where # is the redefined function ID.

If you do not like `xdiff`, you can change the comparison tool by selecting **Fix+Continue > Show Difference > Set Diff Tool** from the menu bar.

Comparing Source Code Files

If you have made several redefinitions to a file, you may need a side-by-side comparison of the entire file. To see how changes to the entire file look, select

Fix+Continue > Show Difference > For File from the menu bar. This opens an `xdiff` window that displays the entire file rather than just the function.

You can get the same comparison results from the Debugger command line if you enter the following command:

```
show_diff -file time1.C
```

Ending the Session

Exit the Debugger by selecting **Admin > Exit** from the menu bar.

Detecting Heap Corruption

The heap is a portion of memory used to support dynamic memory allocation/deallocation via the `malloc` and `free` function. This chapter describes heap corruption detection and covers the following topics:

Typical Heap Corruption Problems

Due to the dynamic nature of allocating and deallocating memory, the heap is vulnerable to the following typical corruption problems:

- *boundary overrun*: a program writes beyond the `malloc` region.
- *boundary underrun*: a program writes in front of the `malloc` region.
- *access to uninitialized memory*: a program attempts to read memory that has not yet been initialized.
- *access to freed memory*: a program attempts to read or write to memory that has been freed.
- *double frees*: a program frees some structure that it had already freed. In such a case, a subsequent reference can pick up a meaningless pointer, causing a segmentation violation.
- *erroneous frees*: a program calls `free`() on addresses that were not returned by `malloc`, such as static, global, or automatic variables, or other invalid expressions. See the `malloc`(3f) man page for more information.

Finding Heap Corruption Errors

To find heap corruption problems, you must relink your executable with the `-lmalloc_ss` library instead of the standard `-lmalloc` library. By default, the `-lmalloc_ss` library catches the following errors:

- `malloc` call failing (returning NULL)
- `realloc` call failing (returning NULL)
- `realloc` call with an address outside the range of heap addresses returned by `malloc` or `memalign`
- `memalign` call with an improper alignment
- `free` call with an address that is improperly aligned
- `free` call with an address outside the range of heap addresses returned by `malloc` or `memalign`

If you also set the `MALLOC_FASTCHK` environment variable, you can catch these errors:

- `free` or `realloc` calls where the words prior to the user block have been corrupted
- `free` or `realloc` calls where the words following the user block have been corrupted
- `free` or `realloc` calls where the address is that of a block that has already been freed. This error may not always be detected if the area around the block is reallocated after it was first `freed`.

Compiling with the Malloc Library

You can compile your executable from scratch as follows:

```
% cc -g -o targetprogram targetprogram.c -lmalloc_ss
```

You can also relink it by using:

```
% ld -o targetprogram targetprogram.o -lmalloc_ss ...
```

An alternative to rebuilding your executable is to use the `_RLD_LIST` environment variable to link the `-lmalloc_ss` library. See the `rld(1)` man page.

Setting Environment Variables

After compiling, invoke the Debugger with your executable as the target. In **Execution View**, you can set environment variables to enable different levels of heap corruption detection from within the `malloc` library, as follows:

`MALLOC_CLEAR_FREE`

Clears data in any memory allocation freed by `free`. It requires that `MALLOC_FASTCHK` be set.

`MALLOC_CLEAR_FREE_PATTERN` *pattern*

Specifies a pattern to clear the data if `MALLOC_CLEAR_FREE` is enabled. The default pattern is `0xcafebeef` for the 32-bit version, and `0xcafebeefcafebeef` for the 64-bit versions. Only full words (double words for 64-bits) are cleared to the pattern.

`MALLOC_CLEAR_MALLOC`

Clears data in any memory allocation returned by `malloc`. It requires that `MALLOC_FASTCHK` be set.

`MALLOC_CLEAR_MALLOC_PATTERN` *pattern*

Specifies a pattern to clear the data if `MALLOC_CLEAR_MALLOC` is enabled. The default pattern is `0xfacebeef` for the 32-bit version, and `0xfacebeeffacebeef` for the 64-bit versions. Only full words (double words for 64-bits) are cleared to the pattern.

`MALLOC_FASTCHK`

Enables additional corruption checks when you call the routines in this library. Error detection is done by allocating a space larger than the requested area, and putting specific patterns in front of and behind the area returned to the caller. When `free` or `realloc` is called on a block, the patterns are checked, and if the area was overwritten, an error message is printed to `stderr` using an internal call to the routine `ssmalloc_error`. Under the Debugger, a trap may be set at exit from this routine to catch the program at the error.

`MALLOC_MAXMALLOC` *n*

Where *n* is an integer in any base, sets a maximum size for any `malloc` or `realloc` allocation. Any request exceeding that size is flagged as an error, and returns a NULL pointer.

`MALLOC_NO_REUSE`

Specifies that no area that has been freed can be reused. With this option enabled, no actual free calls are made and process space and swap requirements can grow quite large.

`MALLOC_TRACING`

Prints out all `malloc` events including address and size of the `malloc` or `free`. When running a trace in the course of a performance experiment, you need not set this variable because running the experiment automatically enables it. If the option is enabled when the program is run independently, and the `MALLOC_VERBOSE` environment variable is set to 2 or greater, trace events and program call stacks are written to `stderr`.

`MALLOC_VERBOSE`

Controls message output. If set to 1, minimal output displays; if set to 2, full output displays.

For further information, see the `malloc_ss(3)` man page.

Trapping Heap Errors Using the Malloc Library

If you are using the `-lmalloc_ss` library, you can use the Trap Manager to set a stop trap at the exit from the function `ssmalloc_error` that is called when an error is detected. Errors are detected only during calls to heap management routines, such as `malloc()` and `free()`. Some kinds of errors, such as overruns, are not detected until the block is `freed` or `realloced`.

When you run the program, the program halts at the stop trap if a heap corruption error is detected. The error and the address are displayed in **Execution View**. You can also examine the **Call Stack** at this point to get stack information. To find the next error, click the **Continue** button.

If you need more information to isolate the error, set a watchpoint trap to detect a `write` at the displayed address. Then rerun your program. Use `MALLOC_CLEAR_FREE` and `MALLOC_CLEAR_MALLOC` to catch problems from attempts to access uninitialized or freed memory.

Note: You can run programs linked with the `-lmalloc_ss` library outside of the Debugger. The trade-off is that you have to browse through the `stderr` messages and catch any errors through visual inspection.

Heap Corruption Detection Tutorial

This tutorial demonstrates how to detect corruption errors by using the `corrupt` program. The `corrupt` program has already been linked with the SpeedShop `malloc` library (`libmalloc_ss`). The `corrupt` program listing is as follows:

```
#include <string.h>
void main (int argc, char **argv)
{
  char *str;
  int **array, *bogus, value;

  /* Let us malloc 3 bytes */
  str = (char *) malloc(strlen(''bad''));

  /* The following statement writes 0 to the 4th byte */
  strcpy(str, ''bad'');

  free (str);

  /* Let us malloc 100 bytes */
  str = (char *) malloc(100);
  array = (int **) str;

  /* Get an uninitialized value */
  bogus = array[0];

  free (str);
  /* The following is a double free */
  free (str);
/* The following statement uses the uninitialized value as a pointer */
   value = *bogus;
}
```

To start the tutorial:

1. Enter the following:

```
% mkdir demos
% mkdir demos/mallocbug
% cd demos/mallocbug
% cp /usr/demos/WorkShop/mallocbug/* .
```

2. Invoke the Debugger by typing:

```
% cvd corrupt &
```

   The Main View window displays with `corrupt` as the target executable.

3. Open the **Execution View** window (if it is minimized) and set the `_SSMALLOC_FASTCHK` and `_SSMALLOC_CLEAR_MALLOC` environment variables.

   If you are using the C shell, type:

```
% setenv _SSMALLOC_FASTCHK
% setenv _SSMALLOC_CLEAR_MALLOC
```

   If you are using the Korn or Bourne shell, type:

```
$ _SSMALLOC_FASTCHK=
$ _SSMALLOC_CLEAR_MALLOC=
$ export _SSMALLOC_FASTCHK _SSMALLOC_CLEAR_MALLOC
```

4. To trap any `malloc` corruption problems, you must enter the following at the `cvd` command line:

```
cvd> set $pendingtraps=true
cvd> stop exit ssmalloc_error
```

   A stop trap is set at the exit from the `malloc` library `ssmalloc_error`.

5. Enter the following at the `cvd` command line:

```
cvd> run
```

   The program executes. Observe **Execution View** as the program executes.

   A heap corruption is detected and the process stops at one of the traps. The type of error and its address display in **Execution View** (see example in Figure 9-1, page 121.)

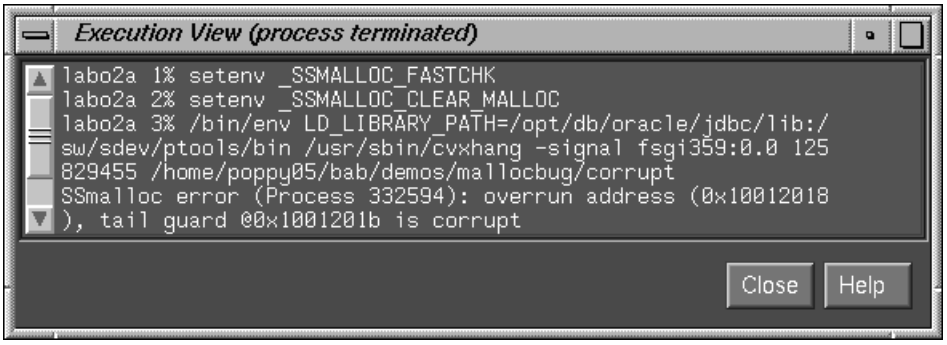

Figure 9-1 Heap Corruption Warning Shown in **Execution View**

6. Select **Views > Call Stack** from the Main View window menu bar.

   Call Stack opens displaying the call stack frame at the time of the error (see Figure 9-2).

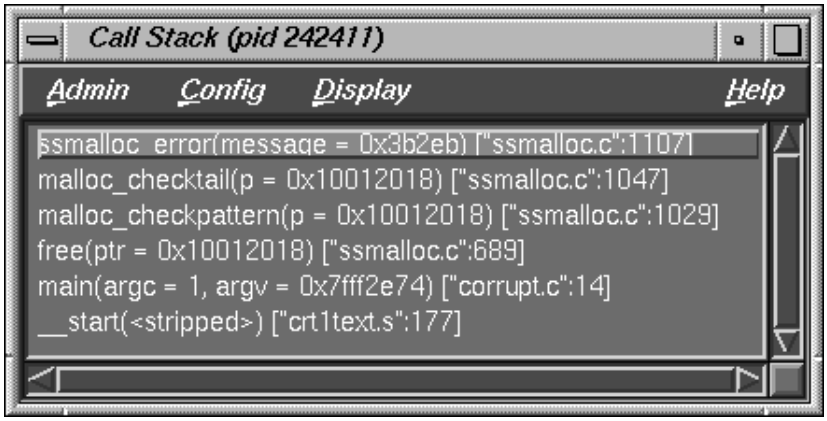

Figure 9-2 Call Stack at Boundary Overrun Warning

7. Click the **Continue** button in the Main View window's control panel. Watch the **Execution View** and **Call Stack** windows.

The process continues from the stop at the boundary overrun warning until it hits the next trap where an erroneous `free` error occurs.

8. Click the **Continue** button again and watch the **Execution View** and **Call Stack** windows.

   This time the process stops at a bus error or segmentation violation. The PC stops at the following statement because `bogus` was set to an uninitialized value:

   ```
   value=*bogus
   ```

9. Enter **`p &bogus`** on the Debugger command line at the bottom of the Main View window.

   This gives us the address for the `bogus` variable and has been done in Figure 9-3, page 123. We need the bad address so that we can set a watchpoint to find out when it is written to. (This example has an address of `0x7fffaef4`; your address will be different.)

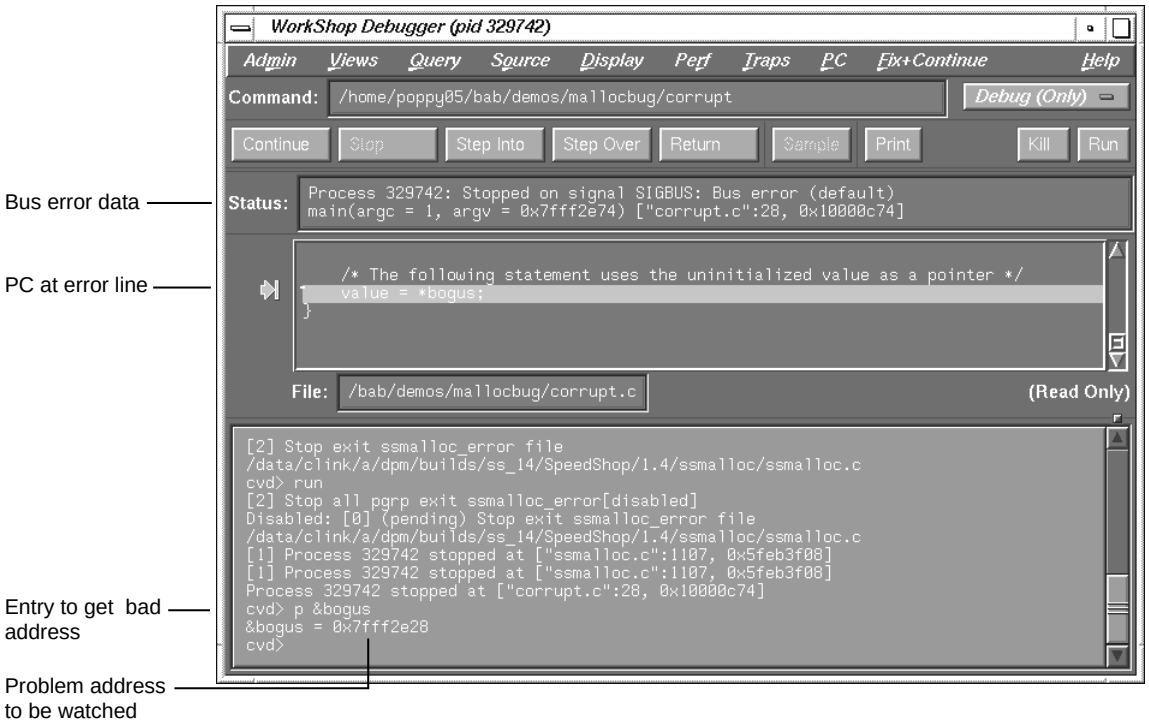

Figure 9-3 Main View at Bus Error

10. Deactivate the stop trap by clicking the toggle button next to the trap description in the **Trap Manager** window, and click the **Kill** button in the Main View window to kill the process.

11. Click on the **Clear** button in the **Trap Manager** window.

12. Type the following command in the **Trap** field. This includes the address you obtained from the Debugger command line (see Figure 9-3, page 123). This sets a watchpoint that is triggered if a `write` is attempted at that address.

Note: Use the address from your system, not the one shown here.

`stop watch address` ***0x7fffaef4*** **`for write`**

13. Click the **Add** button.
14. Click the **Run** button and observe the Main View window.

    The process stops at the point where the `bogus` variable receives a bad value. Details of the error display in the Main View window's **Status** field.

Multiple Process Debugging

The WorkShop Debugger lets you debug threaded applications as well as programs that use multiple processes spawned by `fork` or `sproc`. You can also control a single process or all members of a process group, attach child processes, and specify that spawned processes inherit traps from the parent process. The Trap Manager provides special commands to facilitate debugging multiple processes by setting traps that apply to the entire process group.

The **Multiprocess View** window is for use by C, C++, and Fortran users. If you are debugging Ada code, you should use the **Task View** window available through the **View** menu of the Main View window (see "Task View", page 207).

Currently, **Multiprocess View** handles the following multiple process situations:

- *True multiprocess program*, which refers to a tightly integrated system of `sproc`'d processes, generated by the MIPSpro Automatic Parallelization Option. For more information on parallel processing, see the `auto_p(5)` man page, or the *MIPSpro Automatic Parallelizer Programmer's Guide*.
- *Auto-fork application*, which is a process that spawns a child process and then runs in the background.
- *Fork application*, which is a process that spawns child processes and can interact with them.
- *Locally distributed application*, which is an application that involves two different executables running in different processes on the same host coordinated by a rendezvous mechanism.
- *MPI single system image application*, which is an MPI application that runs on the same host.

This chapter discusses the details of multiprocess debugging in WorkShop and includes the following topics:

- "Debugging a Pthreaded Program", page 140
- "Debugging an MPI Single System Image Application", page 146

Using the Multiprocess View Window

The **Multiprocess View** window is brought up by selecting **Admin > Multiprocess View** from the menu bar of the Main View window.

This window can display individual processes or operate on a process group. By default, a *process group* includes the parent process and all descendants spawned by `sproc`. Processes spawned with `fork` during the session can be added to the process group automatically when they are created. For a program compiled with the MIPSpro Automatic Parallelization Option, a process group includes all threads generated by the option. Any process to which you have read/write access can also be added to the process group. All `sproc`'d processes must be in the same process group, since they share information.

Note: Any child process that performs an `exec` with `setuid` (set user ID) enabled will not become part of the process group.

Each process in the session can have a standard main view window session associated with it. However, all processes in a process group appear on a single **Multiprocess View** window.

When debugging multiprocess applications, you should disable the `SIGTERM` signal by selecting **Views > Signal Panel** from the Main View window menu bar. Although multiprocessing debugging is possible with `SIGTERM` enabled, the multiprocess application may not terminate gracefully after execution is complete.

Starting a Multiprocess Session

The first step in debugging multiple processes is to invoke the Debugger with the parent process. Then select **Admin > Multiprocess View** from the menu bar.

The following figure shows a typical **Multiprocess View** window.

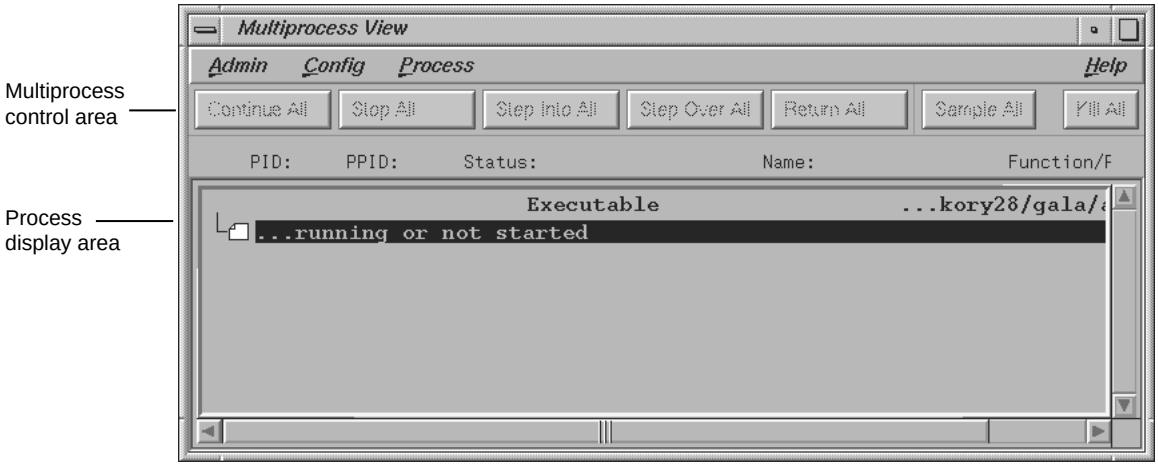

Figure 10-1 Multiprocess View

Viewing Process Status

The process display area of the **Multiprocess View** lists the status of all processes and threads in the process group. For definitions of the various status and states, see "Status of Processes".

To get more information about a process or thread displayed in the process display area, right-click on the process or thread entry. A **Process** menu pops up which is applicable to the selected entry. From this menu you can do the following:

- change entry focus
- create a new window
- focus attention to a user–entered thread
- show thread-specific details
- add or remove an entry

For complete details about the **Process** menu, see "Controlling Multiple Processes", page 206.

Using Multiprocess View Control Buttons

The **Multiprocess View** window uses the same control buttons as the Main View window with the following exceptions:

- Buttons are applied to all processes as a group.
- There are no **Return**, **Print**, or **Run** buttons.

Control buttons in the **Multiprocess View** window have the same effect as clicking the corresponding button in the Main View window of each individual process. For definitions of the buttons, see "Multiprocess View Control Buttons", page 198.

Multiprocess Traps

As discussed in Chapter 5, "Setting Traps", page 65, the trap qualifiers `[all]` and `[pgrp]` are used in multiprocess analysis. The `[all]` entry stops or samples all processes when a trap fires. The `[pgrp]` entry sets the trap in all processes within the process group that contains the trap location. The qualifiers can be entered by default by using the **Stop All Default** and **Group Trap Default** selections, respectively, in the **Traps** menu of Trap Manager. The Trap Manager is brought up from the **Views** menu of the Main View window.

Viewing Pthreaded Applications

The **Multiprocess View** supports a hierarchical view of your pthreaded applications. Select the folder icons of your choosing to get more information about a process or thread.

Perform the following from within the **Multiprocess View** window to get additional information about a process or thread:

1. Double-click on a folder icon.

   The process display expands to show its pthreads.

2. Double-click to select the pthread of your choosing.

   The call stack for that thread displays.

Adding and Removing Processes

To add a process, select **Add** from the **Process** menu. In the **Switch Dialog** dialog window, select one of the listed processes or enter a process ID in the **Process ID** field and click the **OK** button.

To remove a process, click on the process name in the **Multiprocess View** window and select **Remove** from the **Process** menu. Be aware that a process in a `sproc` process group cannot be removed. Likewise, you cannot remove a pthread from a pthread group.

Multiprocess Preferences

The **Preferences** option in the **Config** menu brings up the **Multiprocess View Preferences** dialog. The preferences on this dialog lets you determine when a process is added to the group, specify process behavior, specify the number of call stack levels to display, and so forth.

For details about **Multiprocess View Preference** options, see "Controlling Preferences", page 199.

Bringing up Additional Main View Windows

To create a Main View window for a process, highlight that process in the **Multiprocess View** window. Then, select **Process > Create new window** in the **Multiprocess View** window.

Debugging a Multiprocess C Program

This section uses a C program that generates numbers in the Fibonacci sequence to demonstrate the following tasks when using the debugger to debug multiprocess code:

- Stopping a child process on a `sproc`
- Using the buttons in the **Multiprocess View** window
- Setting traps in the parent process only
- Setting group traps

The `fibo` program uses `sproc` to split off a child process, which in turn uses `sproc` to split off a grandchild process. All three processes generate Fibonacci numbers until stopped. You can find the source for `fibo.c` in the `/usr/demos/WorkShop/mp` directory. A listing of the `fibo.c` source code follows:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/prctl.h>

int NumberToCompute = 100;
int fibonacci();
void run(),run1();

int fibonacci(int n)
{
int f, f_minus_1, f_plus_1;
int i;

      f = 1;
      f_minus_1 = 0;
      i = 0;

      for (; ;) {
          if (i++ == n) return f;
          f_plus_1 = f + f_minus_1;
           f_minus_1 = f;
           f = f_plus_1;
      }
}

void run()
{
int fibon;
      for (; ;) {
          NumberToCompute = (NumberToCompute + 1) % 10;
          fibon = fibonacci(NumberToCompute);
          printf("%d'th fibonacci number is %d\n",
               NumberToCompute, fibon);
      }
}
```

```
void run1()
{
int grandChild;

      errno = 0;
      grandChild = sproc(run,PR_SADDR);

      if (grandChild == -1) {
          perror("SPROC GRANDCHILD");
      }
      else
          printf("grandchild is %d\n", grandChild);
      run();
}

void main ()
{
int second;

      second = sproc(run1,PR_SADDR);
      if (second == -1)
          perror("SPROC CHILD");
      else
          printf("child is %d\n", second);

      run();
      exit(0);
}
```

Launch the Debugger in Multiprocess View

Perform the following to start, compile the program, and run the Debugger:

1. Copy the program source from the demo directory as follows:

   % **cp /usr/demos/WorkShop/mp .**

2. Compile `fibo.c` by entering the following command:

   % **cc -g fibo.c -o fibo**

3. Invoke the Debugger on `fibo` as follows:

   ```
   % cvd fibo &
   ```

4. Call up the **Multiprocess View** by selecting **Admin > Multiprocess View** from the Main View menu bar.

The next section uses the `fibo` program to illustrate some of the functionality of the Multiprocess window.

Using Multiprocess View to Control Execution

To examine each process as it is created, you must set preferences so that each child process created stops immediately after being created. The following steps show how this can be done:

1. Select **Config > Preferences** from the menu bar in the **Multiprocess View** window.
2. Toggle off **Resume child after attach on sproc** in the **Multiprocess View Preferences** window.
3. Toggle off **Copy traps to sproc'd processes** so you can experiment with setting traps later.
4. Click on the **OK** button to accept the changes.
5. Click on the **Run** button in the Main View window to execute the `fibo` program.

   Watch the **Multiprocess View** window, you will see the main process appear and spawn a child process, which stops as soon as it appears. This is because you turned off the **Resume child after attach on sproc** option. Notice also that the Main View window switched to the stopped child process.

6. Click on the **Stop** button in the **Multiprocess View** window.

   The control buttons on the **Multiprocess View** window may be used to control all processes simultaneously, or the control buttons on any Main View window may be used to control that individual process separately.

7. Click on the first line (that is, the main process) in the process pane of the **Multiprocess View** window to highlight this line.
8. Select **Process > Create new window** from the menu bar of the **Multiprocess View** window.

A new Main View window displays with a debug session for the main process.

Note: You may get a warning that **.../write.s is missing**. This refers to assembly code and can be ignored. The new Main View window will not have source in its source pane.

---

9. Select **Views > Call Stack** from the menu bar of the Main View window you just created to create a **Call Stack** window.
10. Double-click on the line in the **Call Stack** window that contains `run ()`. This brings up the `fibo.c` source for the main process in the Main View window.
11. Select **Admin > Close** from within the **Call Stack** window to close it.
12. Click on **Cont** in the **Multiprocess View** window. The first child, created in Step 5, now spawns a grandchild process that stops in `_nsproc`, as shown here:

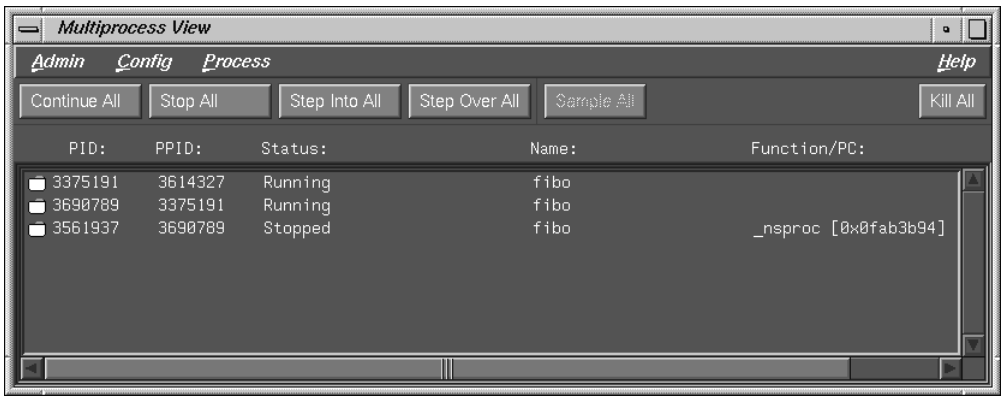

Figure 10-2 Examining Process State Using **Multiprocess View**

13. A Main View window switches to the new stopped process. Click on **Stop** in the **Multiprocess View** window.
14. Repeat steps 7 through 11 to bring up a Main View window for the parent process.

Using the Trap Manager to Control Trap Inheritance

The instructions in this section assume that you have just run the tutorial in "Using Multiprocess View to Control Execution", page 132.

This section shows you how to use the Trap Manager to set traps that affect one or all processes in the `fibo` process group. For complete information on using the Trap Manager, refer to Chapter 5, "Setting Traps", page 65.

1. Select **Views > Trap Manager** from the Main View window for the parent process. Traps are specific to the processes in the Main View window in which they are set.
2. Select **Display > Show Line Numbers** (from the same Main View window) to turn on line numbering in the source pane, if not already showing.
3. Click to the left of line `32`, to set a breakpoint/stop trap for the parent process. Line `32` reads as follows:

   ```
   32 Number to Compute = (NumberToCompute + 1) % 10
   ```

   Line 32 highlights in red to indicate that a breakpoint has been set. A corresponding trap command appears in the `Trap` text box in the **Trap Manager** window; and the trap is added to the list on the **Active Traps** list of the same window. Remember, this trap will affect only the parent process.
4. Click on the **Cont** button in the **Multiprocess View** window. The parent process has stopped, but the other processes are probably still running.
5. Insert the word `pgrp` (that is, “process group”) after the word `stop` in the **Trap** field of the **Trap Manager** window.

   The trap should now read `Stop pgrp at ...`. As the command suggests, `pgrp` affects the whole process group.
6. Click on the **Modify** button.

   The trap now affects two child processes. Watch the **Multiprocess View** window to see the running processes in the process group stop at the trap on line 32.
7. Select **Traps > Group Trap Default** from the **Trap Manager** window. Any additional traps that you set using the **Trap Manager** affect the entire process group. Any previously set traps are not be affected.

8. Select the text of line **23**, found in the source pane of the Main View window associated with the parent process. This line reads as follows:

   ```
   23 f_minus_1 = f;
   ```

9. Select **Traps > At Source Line** from the menu bar of the **Trap Manager** window. The trap you have just set includes the modifier `pgrp`.

10. Select **Admin > Exit** from any Main View window to close your session and end this tutorial.

Debugging a Multiprocess Fortran Program

The section of this chapter presents a few standard techniques to assist you in debugging a parallel program. This section shows you how to debug the sample program.

See also Chapter 2, "Basic Debugger Usage", page 9 for important related information.

General Fortran Debugging Hints

Debugging a multiprocessed program is more involved than debugging a single-processor program. Therefore, you should debug a single-processor version of your program first and try to isolate the problem to a single parallel `DO` loop.

After you have isolated the problem to a specific `DO` loop, change the order of iterations in a single-processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If it cannot be multiprocessed, you will see that changing the order in which the loops execute will cause the single-processor version to produce wrong answers. If wrong answers are produced, you can use standard single-process debugging techniques to find the problem. (See Chapter 2, "Basic Debugger Usage", page 9 for important related information.)

If this technique fails, you must debug the multiprocessed version. To do this, compile your code with the `-g` and `-FLIST:=ON` flags. The `-FLIST:=ON` flags save the file containing the multiprocessed `DO` loop Fortran code in a file called `total.w2f.f` and a file `tital.rii` and an `rii_files` directory.

Fortran Multiprocess Debugging Session

This section shows you how to debug a small segment of multiprocessed code. The source code for this tutorial, `total.f`, can be found in the directory `/usr/demos/WorkShop/mp`.

A listing of this code is as follows:

```
program driver
      implicit none
      integer iold(100,10), inew(100,10),i,j
      double precision aggregate(100, 10),result
      common /work/ aggregate
      result=0.
      call total(100, 10, iold, inew)
      do 20 j=1,10
        do 10 i=1,100
          result=result+aggregate(i,j)
10    continue
20  continue
      write(6,*)' result=',result
      stop
      end

      subroutine total(n, m, iold, inew)
      implicit none
      integer n, m
      integer iold(n,m), inew(n,m)
      double precision aggregate(100, 100)
      common /work/ aggregate
      integer i, j, num, ii, jj
      double precision tmp

      C$DOACROSS LOCAL(i,ii,j,jj,num)
      do j = 2, m-1
        do i = 2, n-1
          num = 1
          if (iold(i,j) .eq. 0) then
            inew(i,j) = 1
          else
          num = iold(i-1,j) +iold(i,j-1) + iold(i-1,j-1) +
&         iold(i+1,j) + iold(i,j+1) + iold(i+1,j+1)
```

```
        if (num .ge. 2) then
          inew(i,j) = iold(i,j) + 1
        else
          inew(i,j) = max(iold(i,j)-1, 0)
        end if
      end if
      ii = i/10 + 1
      jj = j/10 + 1
      aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
    end do
  end do
  end
```

In the program, the local variables are properly declared. The `inew` always appears with `j` as its second index, so it can be a share variable when multiprocessing the `j` loop. The `iold`, `m`, and `n` are only read (not written), so they are safe. The problem is with `aggregate`. The person analyzing this code deduces that, because `j` is always different in each iteration, `j/10` will also be different. Unfortunately, since `j/10` uses integer division, it often gives the same results for different values of `j`.

While this is a fairly simple error, it is not easy to see. When run on a single processor, the program always gets the right answer. Sometimes it gets the right answer when multiprocessing. The error occurs only when different processes attempt to load from and/or store into the same location in the `aggregate` array at exactly the same time.

Debugging Procedure

Perform the following to debug this code:

1. Create a new directory for this exercise:

   ```
   % mkdir demos/mp
   ```

2. `cd` to the new directory and copy the following program source into it:

   ```
   % cp /usr/demos/WorkShop/mp .
   ```

3. Edit the `total.f` file in a shell editor, such as `vi`:

   ```
   % vi total.f
   ```

4. Reverse the order of the iterations for demonstration purposes.

   Replace

```
do j = 2, m-1
```

with

```
do j = m-1, 2, -1
```

This still produces the right answer with one process running, but the wrong answer when running with multiple processes. The local variables look right, there are no equivalence statements, and `inew` uses only simple indexing. The likely item to check is `aggregate`. Your next step is to look at `aggregate` with the Debugger.

5. Compile the program with `-g` option as follows:

```
% f77 -g -mp total.f -o total
```

6. If your debugging session is not running on a multiprocessor machine, you can force the creation of two threads, for example purposes, by setting an environment variable. If you use the C shell, type:

```
% setenv MP_SET_NUMTHREADS 2
```

Is you use the Korn or Bourne shell, type:

```
$ MP_SET_NUMTHREADS=2
$ export MP_SET_NUMTHREADS
```

7. Enter the following to start the Debugger:

```
% cvd total &
```

The Main View window displays.

8. Select **Display > Show Line Numbers** from the Main View menu bar to show the line numbers.

9. Select **Source > Go To Line** from the Main View menu bar.

And enter **44**.

Line 44 is as follows:

```
aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
```

10. You will now set a stop trap at this line, so you can see what each thread is doing with `aggregate`, `ii`, and `jj`. You want this trap to affect all threads of the process group. One way to do this is to turn on trap inheritance in the

Multiprocess View Preferences dialog. To open this dialog, select **> Admin > Multiprocess View** from the Main View menu bar to open the **Multiprocess View** window.

Then, select **Config > Preferences** from within the **Multiprocess View** window.

Another way is to use the Trap Manager to specify group traps, as follows.

a. Select **Views > Trap Manager** from the Main View window menu bar to open the Trap Manager.

b. Select **Traps > Group Trap Default** from the **Trap Manager** window.

11. Click-drag to select line 44 in the Main View window.

12. Open the **Trap Manager** window from the Main View window menu bar by using **Views > Trap Manager**.

    Then select **Traps > At Source Line** from the **Trap Manager** window.

    This sets a stop trap that reads as follows in the `cvd` pane of the Main View window:

    ```
    Stop pgrp at file /usr/demos/WorkShop/mp/total.f line 44
    ```

13. Select **Admin > Multiprocess View** from the menu bar in the Main View window to monitor status of the two processes.

    You are now ready to run the program.

14. Click the **Run** button in the Main View window.

    As you watch the **Multiprocess View**, you see the two processes appear, run, and stop in the function `_mpdo_total_1`. It is unclear, however, if the Main View window is now relative to the master process, or that it has switched to the slave process.

15. Right-click on the name of the slave process in the **Multiprocess View** window and select **Process > Create a new window**.

    A new window is displayed that launches a debug session for the process. Now, both master and slave processes should display in respective Main View windows.

16. Invoke the Variable Browser as follows from the Menu Bar of each process: **Views > Variable Browser**.

17. Look at `ii` and `jj` in the following figure.

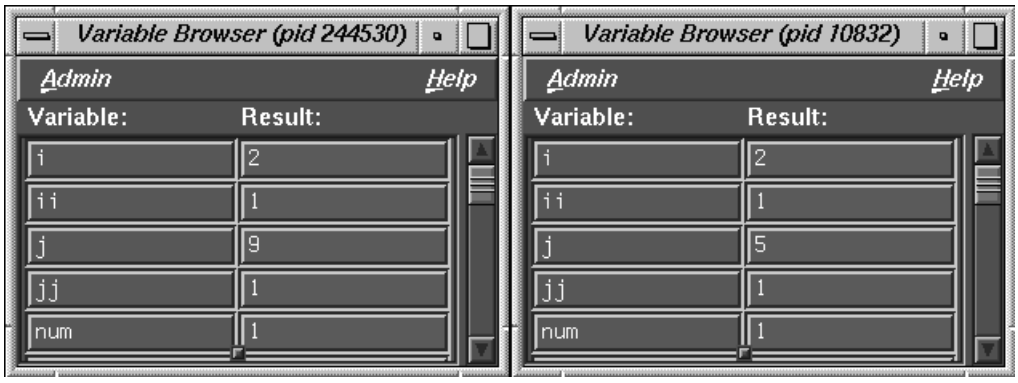

Figure 10-3 Comparing Variable Values from Two Processes

They have the same values in each process; therefore, both processes may attempt to write to the same member of the array `aggregate` at the same time. So `aggregate` should not be declared as a share variable. You have found the bug in your parallel Fortran program.

Debugging a Pthreaded Program

`cvd` supports the debugging of IRIX 6.4 and 6.5 pthreads. You can view pthread creation and execution through the **Multiprocess View** window. Through this window you can:

- View a hierarchal display of a threaded application
- View a process/pthread relationship
- Expand individual call stacks

C, C++, and Fortran users should use the **Multiprocess View** window when debugging pthreads. Ada users should use the **Task View** window.

The next sections give hints on debugging pthreaded programs, list differences between 6.4 and 6.5 threads, and illustrate how to debug a program that uses IRIX 6.5 pthreads.

User-Level Continue of Single 6.5 POSIX Pthread

The ability to "continue" or "free run" a single POSIX pthread under IRIX 6.5 is available at the user level with WorkShop release 2.8. However, use of this new debugging feature can, in certain specific circumstances, lead to anomalous and possibly confusing behavior. Such behavior occurs when the single thread that is continued or free run encounters either a "blocking" or "scheduling" situation in the operating system or the pthreads library.

When such situations arise, the operating system (or, in some cases, the pthreads library) must take action to dispose of the single continued or free run thread and, possibly, newly created threads. In the course of this action the debugging user will see things occur, with both the single continued or free run thread as well as all other threads, that are confusing because complex thread scheduling algorithms are invoked by both the operating system and the pthreads library to recover from the original blocking or scheduling incident. Debugging true POSIX pthreads is difficult, and users of this new feature, allowing a continue or free run of a single 6.5 POSIX pthread, will gain even more appreciation of this fact.

This feature has been used internally for some time by the WorkShop debugger. The continue or free run of a single 6.5 pthread is used each time a user requests a single thread step-over of a function. The single thread is allowed to free run through the function which is being stepped over. Thus, if any blocking or scheduling situations occur in the course of this stepping over and associated free run of a single thread, then anomalous behavior can, and does, occur. This is described in the following subsections.

Scheduling Anomalies

Scheduling anomalies *may* occur when the single 6.5 POSIX pthread which is being continued or free run creates a new pthread via a call to the `pthread_create` routine. At the time of the call to `pthread_create` the OS kernel and the pthreads library get into a complex algorithm in deciding how to create the new child pthread. An actual OS kernel thread (OS kernel threads are not available at the user level — they differ from the user level pthread) must be either created anew or found elsewhere to support the user's new child pthread.

First, assume the OS kernel thread is to be found elsewhere, depending on a vast number of things (for example, number of CPUs, environment variables, and so on). The OS kernel *may* (this is very non-deterministic) decide to just put the child pthread on a ready queue, in need of an OS kernel thread. Thus the child does nothing immediately.

Meanwhile, if the parent pthread (via a call to the `pthread_cond_wait` routine) monitors the child pthread's struggle for life, it (the parent) gets parked on a mutex (mutual exchange lock) because the child obviously has not been created yet; it is on the ready queue.

The parent pthread's OS kernel thread becomes available, which causes the OS scheduler to check for work for this newly freed OS kernel thread. It finds the child sitting in the ready queue and assigns the parent's OS kernel thread to the new child pthread. The child then runs to completion and releases its (parent's old) OS kernel thread. The parent, checking for the child's new life via `pthread_cond_wait`, now recaptures its OS kernel thread and things appear to work correctly.

Now, assume the OS kernel thread required by the new child pthread must be created anew. The child is not placed on the 'ready queue'. Again, this is a non-deterministic decision which depends on a large number of variables (number of CPUs, and so on). The OS kernel creates a new OS kernel thread for the child pthread and "engages" it (the child) to that new OS kernel thread.

However, "marriage" of the OS kernel thread and the new child pthread cannot occur until the new OS kernel thread actually runs. This never occurs because, in allowing the single parent 6.5 pthread to continue or run free, it was requested that only one user pthread be run — the parent.

If the parent, however, is using `pthread_cond_wait` to monitor the new life of its child, then it (the parent) is parked on a mutex waiting for the child to run. The parent awaits the child but the child cannot run because only one pthread, the parent, has been requested to run. The debugger displays "running" as the overall status and this is because no events of interest are occurring. Everything is waiting on everything else. Things are not working.

Blocking Anomalies

Blocking anomalies occur when the single 6.5 POSIX thread which is being continued or free run encounters a blocking condition in the course of its running. Blocking has three distinct types:

- Blocking syscalls in the OS kernel (see "Blocking Kernel System Calls", page 326, for a list). When one of these kernel syscalls is blocked by another thread's usage, the OS kernel decides what the next move is regarding the OS kernel thread attached to the user pthread making the call. Control could just transfer to another application, to disk I/O, or whatever.

These syscalls are all I/O-related. The OS kernel thread is, in effect, "blocked", and it is immediately available for reassignment. The best example of a blocking kernel syscall is `writev`, which is used by the common library routine `printf`.

- Various lock blocking in the pthread library, such as mutex (mutual exchange lock). This occurs in user space (`libc`, user code, and so on). The pthread library senses that a pthread is going to block due to another pthread's usage. Control transfers to the `usync_control` routine, which eventually calls a blocking kernel syscall (see the preceding item in this list). Again, the OS kernel decides the fate of the associated OS kernel thread. Unexpected things could start running.
- Other lock blocking in the pthread library, whereby the pthread library senses that a user pthread is going to block but does not go off to `usync_control`. Instead it goes to the `pthread_scheduler` in the pthread library for the disposition of the associated OS kernel thread. The `pthread_scheduler` then reassigns the associated OS kernel thread to another user pthread and unexpected things could start running.

How to Continue a Single POSIX 6.5 Pthread

To continue (or free run) a single POSIX 6.5 pthread, simply click on the **Continue** button in the Main View window. Note that this is different from the function of the **Continue** button in the **Multiprocess View** window, which continues all threads.

Other Pthread Debugging Hints

Observe the following guidelines when debugging pthreaded programs:

- Be aware that the `cvmain` (**Main View**) for release 2.8 (and later) contains options (such as **Continue**, **StepOver**, **StepInto**, and **Return**) that are for a single 6.5 pthread — the pthread that is displayed, or the *focus thread*. Do not use the **Main View** options unless you intend to use them for a single thread.
- C++ exception handling works per process not per thread.
- Using the `step over` function on a `pthread_exit` may produce unexpected results.
- Use **Multiprocess View** not **Task View**.
- Use the WorkShop tools instead of `dbx` for 6.5 pthread debugging whenever possible.

- Do not do a **Next** of `printf`.

Differences between 6.4 and 6.5 Pthreads

The following differences exist between the IRIX 6.4 and the IRIX 6.5 implementation of pthreads:

- In the IRIX 6.5 implementation, many user threads are serviced by a few kernel micro-threads. This can result in data race problems that may occur as kernel micro-threads switch from one user pthread to another.

  You can alleviate this problem with the following command:

  ```
  % ccall pthread_setconcurrency n
  ```

  n is the number of kernel micro-threads made available to service user threads. In this way, you can increase the number of kernel micro-threads available. However, you must not specify a value for *n* that is greater than the number of physical CPUs on your system.

- The **Multiprocess View** window shows different sets of threads or processes depending on which version of IRIX you are running. (This is because in IRIX releases before IRIX 6.5, threads are handled as sprocs, not as threads.)

Pthread Debugging Session

Pthread debugging is highly variable not only from environment to environment but also from IRIX release to IRIX release. Because of this, it is not possible to provide a representative pthread debugging tutorial that can be used by all users.

See "User-Level Continue of Single 6.5 POSIX Pthread", page 141, for an in-depth description of current pthread implementation in IRIX.

Using StepOver of Function Calls on IRIX 6.5+ Systems

When debugging IRIX 6.5 (or greater) pthreads, if you attempt to 'step over' a function call, there is a possibility that pthreads will *block*. This blocking can occur if you attempt to step-over either a direct or indirect call to one of the following:

- One of several blocking pthread library routines (see "Blocking pthreads Library Routines", page 145)

- One of several blocking kernel syscalls (see "Blocking Kernel System Calls", page 326, for a list of the syscalls).

If a pthread does block in either of these situations, an internal breakpoint is reached at `_SGIPT_sched_block` (for blocking pthread library routines) or `_SGIPT_libc_blocking` (for blocking kernel syscalls).

Without these internal breakpoints, when a pthread blocks, control is returned to the OS kernel, at which point any number of events could occur, including a recycling of the kernel micro-thread attached to the user pthread. This might allow another user pthread to resume execution, thereby causing the debugger to appear to be running or appear to be hung because the original thread which blocked is not allowed to run to its return point (since it had its microthread swapped out underneath it).

The OS kernel uses complex algorithms to determine what action to take when a pthread blocks. The debugger's use of the internal breakpoints allows you to take back a degree of control over these complex algorithms by deciding what to do with a thread that has blocked in either `_SGIPT_sched_block` or `_SGIPT_libc_blocking`.

Usually you can simply use **Continue All Pthreads** to release the blocking condition or continue a different individual pthread (different from the one that blocked).

Blocking Kernal Syscall Routines

For OS level 6.5 pthreads, the `Libpthread` entry point `_SGIPT_libc_blocking` is entered when a specific pthread blocks in a kernel syscall. See "Blocking Kernel System Calls", page 326, for a list of these syscalls.

There are many library routines that can call one of these blocking system calls; it is impossible to list all such routines which utilize a blocking system call. Users must be knowledgeable enough to know that if, for example, they call the library routine `printf`, it eventually calls `writev()` which is a blocking system call and thus *may* block.

Blocking pthreads Library Routines

For OS level 6.5 pthreads, the `Libpthread` entry point `_SGIPT_sched_block` is entered when a specific pthread blocks in the pthread library. The following routines are known to block:

- `pthread_cond_wait()`

- pthread_cond_timedwait()
- pthread_mutex_lock()
- pthread_join()
- pthread_exit()
- pthread_rwlock_rdlock()
- pthread_rwlock_wrlock()
- sem_wait()

Debugging an MPI Single System Image Application

cvd supports the debugging of a single system image MPI application. The debugging session is set up so that initially mpirun is being debugged. The following is the typical command line used to invoke cvd on an MPI application:

% **cvd mpirun -args -np 2** ***MPI_app_name***

This example command line indicates that the arguments -np 2 *MPI_app_name* will be passed to mpirun and that cvd will initially be focused on the mpirun process.

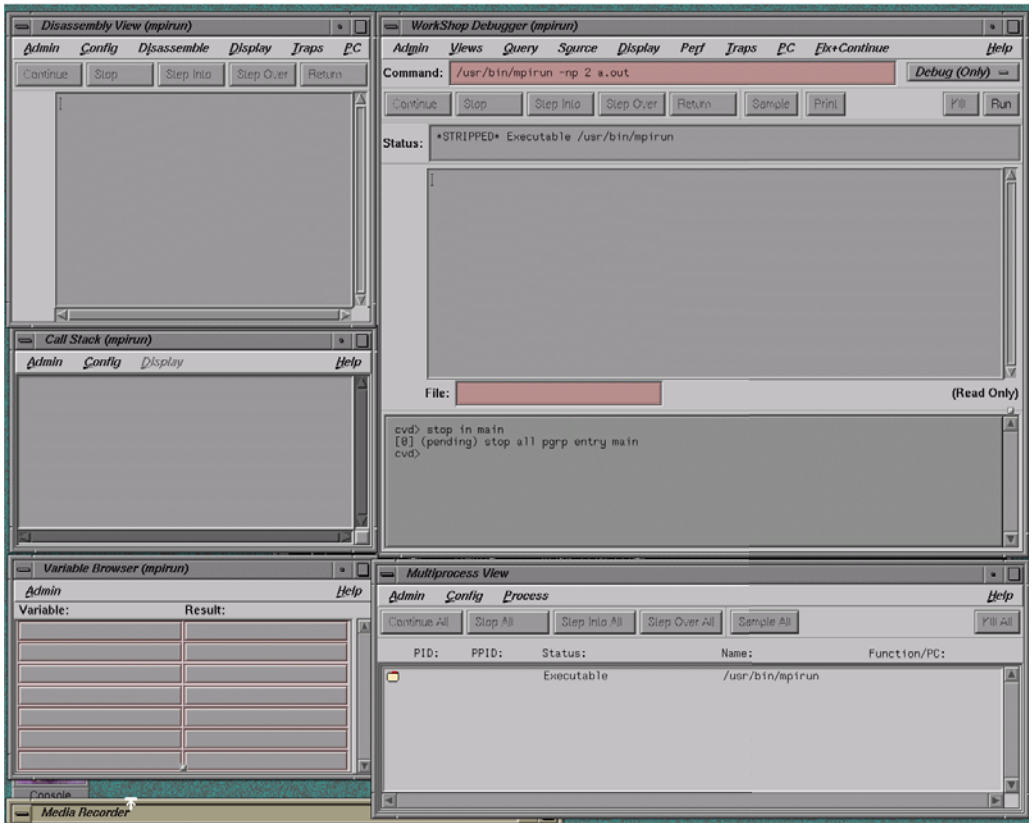

Figure 10-4 Initial MPI start up screen display

An entry point into the MPI application can be used to set a pending trap (breakpoint) in the MPI application. This breakpoint will be resolved when the **Run** button is activated and the actual MPI application is running. If the breakpoint target is valid, the MPI application will stop at the breakpoint and further debugging can be done.

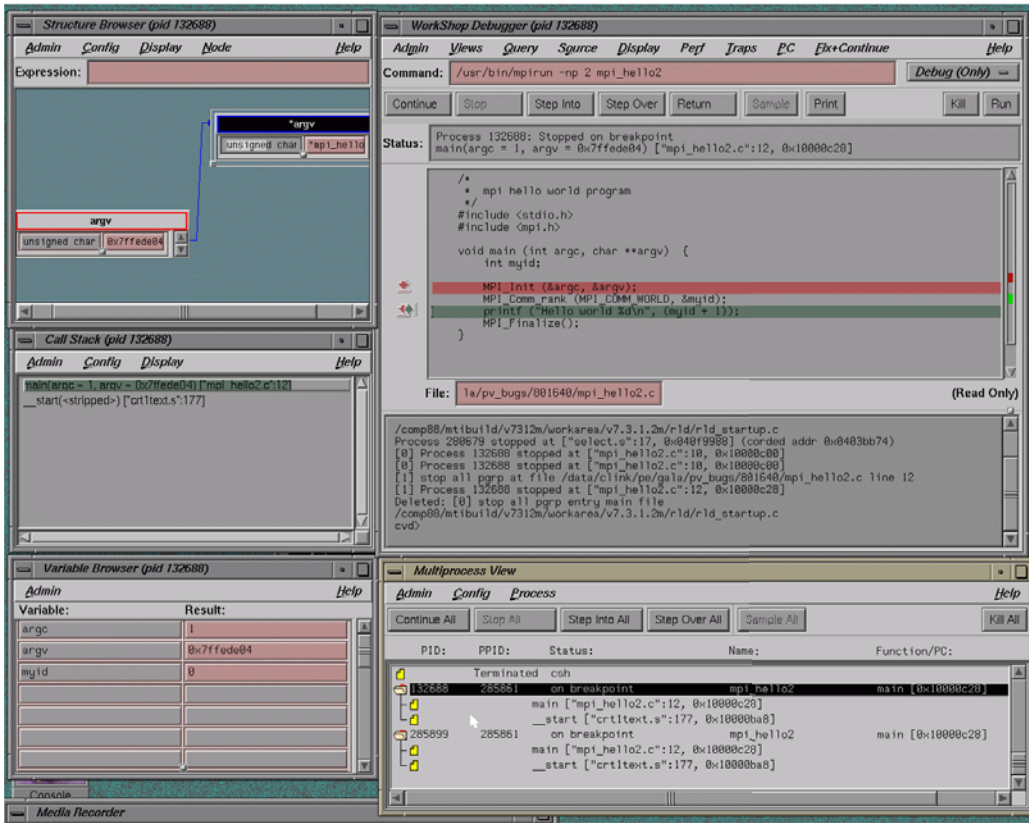

Figure 10-5 Reaching the MPI application breakpoint screen display

The use of the **Multiprocess View** for debugging MPI applications is very similar to what has been presented in the previous sections of this chapter.

The current implementation does not filter out other processes created by login shells so some extra processes may be shown in the **Multiprocess View**.

X/Motif Analyzer

This chapter provides an introduction to the X/Motif Analyzer as well as a tutorial to demonstrate most of the Analyzer functions. Motif is a library of routines that enable you to create user interfaces in an X-environment. The Motif libraries handle most of the low-level event handling tasks common to any GUI system. This way, you can create sophisticated interfaces without having to contend with all of the complexity of X.

The X/Motif Analyzer helps you debug code that calls Motif library routines. The X/Motif Analyzer is integrated with the Debugger so you can issue X/Motif Analyzer commands graphically. To access X/Motif analyzer subwindow select the following from the Main View window menu bar: **Views > X/Motif Analyzer**.

Introduction to the X/Motif Analyzer

The Analyzer contains X/Motif objects (for example, widgets and X graphics contexts) that can be difficult or impossible to inspect through ordinary debugging procedures. It also allows you to set widget-level breakpoints and collect X–event history information in the same manner as using . See the `xscope`(1) man page for more information.`xscope`(blank)

Examiners Overview

When the **X/Motif Analyzer** first displays, it is set to examine widgets. At this point, the window may be blank, or it may display a widget found in the call stack of a stopped process.

At the bottom of the **X/Motif Analyzer** window is a tab panel that shows the current set of examiners. In addition to this tab, the **Widget** Examiner, **Breakpoints**, **Trace**, and **Tree** Examiners tabs are at the bottom of the window. These four tabs are always present. Other examiners are available from the **Examine** menu of the **X/Motif Analyzer** window.

Some examiners cannot be manually selected. They appear only when appropriate to the call stack context. For example, the **Callback** Examiner appears only when a process is stopped somewhere in a widget callback.

Examiners and Selections

If you select text in one examiner and then choose another examiner by using the **Examine** menu, the new examiner is brought up and the text is used as an expression for it. If you selected text that is an inappropriate object for the new examiner, an error is generated.

Alternatively, you can select text, pull down the **Examine** menu, and choose **Selection**. Here, the X/Motif Analyzer attempts to select an appropriate examiner for the text. If the type of text is unknown, the following message displays:

```
Couldn't examine selection in more detail
```

Otherwise, the appropriate examiner is chosen and the text is evaluated.

You can also accomplish this by triple-clicking on a line of text. If the type of text is unknown, nothing happens. Otherwise, the appropriate examiner is chosen and the text is evaluated.

Inspecting Data

X/Motif applications consist of collections of objects (that is, Motif widgets) and make extensive use of X resources such as windows, graphics context, and so on. The construction model of an X window system hinders you from inspecting the internal structures of widgets and X resources because you are presented with ID values. The X/Motif Analyzer lets you to see the data structures behind the ID values.

Inspecting the Control Flow

Traditional debuggers enable you to set breakpoints only in source lines or functions. With the X/Motif Analyzer, you can set breakpoints for specific widgets or widget classes, for specific control flow constructs like callbacks or event handlers, and for specific X events or requests.

Tracing the Execution

The X/Motif Analyzer can trace `Xlib`-level server events and client requests, `Xt`-level event dispatching information, widget life cycle, and widget status information.

Restrictions and Limitations

The X/Motif Analyzer has the following restrictions and limitations:

- The **Breakpoints** Examiner is active only after you have stopped a process and if you have changed `$LD_LIBRARY_PATH`. See "Launching the X/Motif Analyzer", page 152 for more information regarding the correct `$LD_LIBRARY_PATH`.
- Sometimes, gadget names may be unavailable and are displayed as `<object>`. You can minimize this condition by first loading the widget tree.
- `editres` requests (such as, widget selection and widget tree) work only if the process is running or if the process is stopped outside of a system call. This can be annoying when the process is stopped in `select()`, waiting for an X server event.
- The process state and appearance of the Main View window flickers while the X/Motif Analyzer tries to complete an `editres` request when the process is stopped.
- `editres` requests may be unreliable if the process is stopped.

X/Motif Analyzer Tutorial

This section illustrates several features of the X/Motif Analyzer. The demo files in the `/usr/demos/WorkShop/bounce` directory are used to demonstrate the debugging of a running X-application. These files contain the complete C++ source code for the `bounce` program.

This section includes the following subsections:

Setting up the Sample Session

Perform the following to prepare for this session:

1. Enter the following commands:

```
% mkdir demos/bounce
% cd demos/bounce
% cp /usr/demos/WorkShop/bounce .
% make clean
% make bounce
% cvd bounce &
```

   The Debugger is launched, from which you can use the X/Motif Analyzer. Upon invocation, you see the **Execution View** icon and the Main View window.

2. Double-click the **Execution View** icon to open the window. Then, tile your windows so you can clearly see all windows.

3. Click on the **Run** button in the Main View window to run the `bounce` program.

   The **Execution View** window will update with the command that `cvd` is executing.

4. Click **Run** in the **Bounce** window. You will see no action until you have finished the next steps.

5. Select **Actors > Add Red Ball** from the menu bar of the **Bounce** program window.

6. Click on the **Kill** button in the Main View window to terminate the process that has been running.

7. The **Execution View** shows the program output.

Launching the X/Motif Analyzer

Once the `bounce` fileset is built and the debugger is active, you need to launch the X/Motif Analyzer as follows:

1. Select **Views > X/Motif Analyzer** from the menu bar of the Main View window.

2. Click **OK** when asked if you want to change your `$LD_LIBRARY_PATH` environment variables to include `.../usr/lib/WorkShop/Motif`. There are no instrumented MIPS/ABI versions of the libraries.

This includes instrumented versions of the SGI libraries `Xlib`, `Xt`, and `Xm`. These libraries provide debugging symbols and special support for the X/Motif Analyzer. You are now ready to begin the sample session.

Note: Follow the steps in this tutorial precisely as written.

Navigating the Widget Structure

When the X/Motif Analyzer is launched, it brings up the **X/Motif Analyzer** window with an empty **Widget** Examiner tab panel. The tab panels also show the **Breakpoints**, **Trace**, and **Tree** Examiner tab panels (see Figure 11-1, page 154).

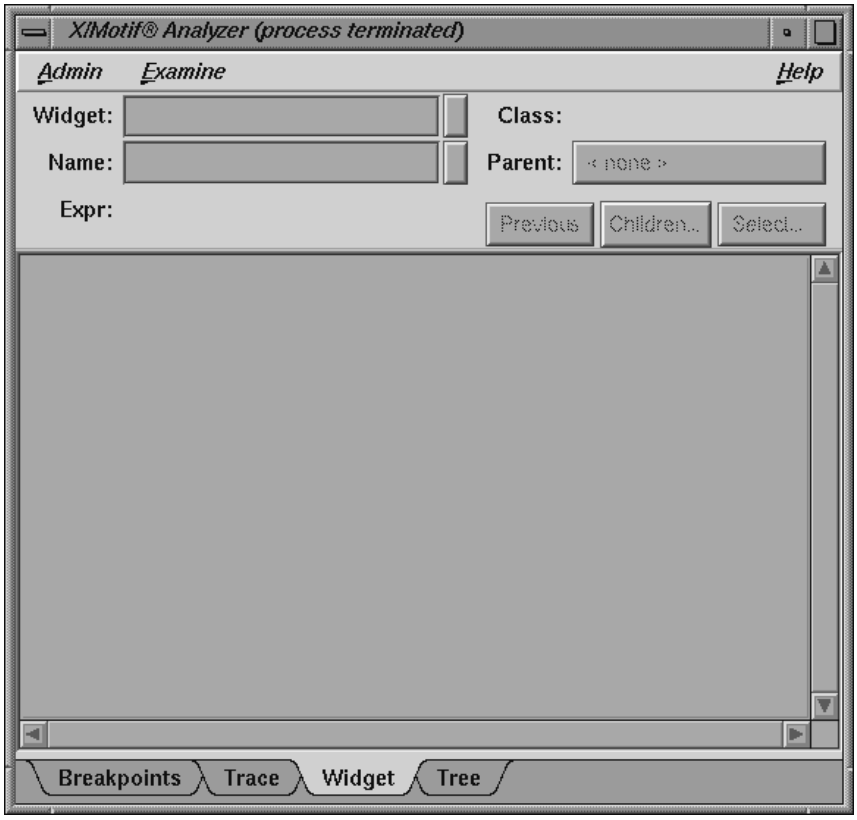

Figure 11-1 First View of the **X/Motif Analyzer** (**Widget** Examiner)

1. Widen the X/Motif Analyzer window shown in Figure 11-1, page 154. This will make it easier to understand what you will be asked to do in this tutorial.
2. Click **Run** in the Main View window to re-run the `bounce` program.

   The instrumented versions of the Motif libraries will now be used.
3. When the **Bounce** window appears, re-size it to make it taller.
4. Click **Run** in the `Bounce` window. You will not see any action until you perform the next steps.
5. Position the **X/Motif Analyzer** and **Bounce** windows side-by-side.

6. Click on the **Select** button in the **X/Motif Analyzer** window.

   This brings up an information dialog and changes the cursor to a plus sign (+). Do not click on the **OK** button in this dialog.

7. Select the `Step` widget by clicking on the **Step** button in the **Bounce** window with the (+) cursor, as described in the **cvmotif** information dialog.

   The Widget Examiner displays the Step widget structure.

8. Click the **Tree** tab in the X/Motif window.

   The **Tree** Examiner panel displays the widget hierarchy of the target object (see Figure 11-2, page 155).

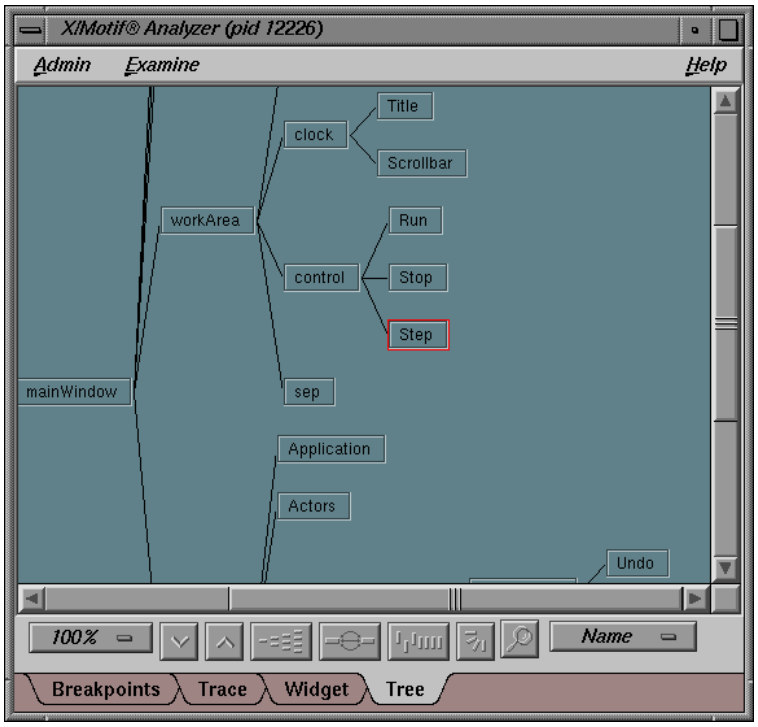

Figure 11-2 Widget Hierarchy Displayed by the Tree Examiner

9. Double-click the **Run** node in the tree. (**Run** is in the upper-right area of the window).

   This brings up the **Widget Examiner** that displays the `Run` widget structure. Notice that the **parent** text area displays the name of the current widget's parent, which is **control**.

10. Click on the word **control** displayed on the **Parent** button at the top of the Widget Examiner in the X/Motif Analyzer window.

    This switches the view to the `Run` widget's parent, the `control` object, as shown in the **Name** field. And, the Widget Examiner displays the **Control** widget structure.

    You can now navigate through the widget hierarchy using either the **Widget** Examiner or the **Tree** Examiner.

Examining Widgets

1. In the Widget Examiner, click on the **Children** button to see the menu, and select **Run** from that menu.

   The `Run` widget structure displays in the examiner.

2. Select **Actors > Add Red Ball** from the **Bounce** window. You should see a bouncing red ball.

3. Enter **`stop in Clock::timeout`** at the `cvd` command line in the Main View window.

   After you press **Enter**, the red ball will stop bouncing.

4. Select **Continue** in the Main View window a few times to observe the behavior of `bounce` with this breakpoint added.

5. Select the **Breakpoints** tab in the **X/Motif Analyzer** window. This calls up the **Breakpoints Examiner** which allows you to set widget-level breakpoints.

6. In the **Callback Name** text field, enter **`activateCallback`**, then click on the **Add** button to add a breakpoint for the `activateCallback` object of the **Run** button widget. The result is displayed in Figure 11-3, page 157.

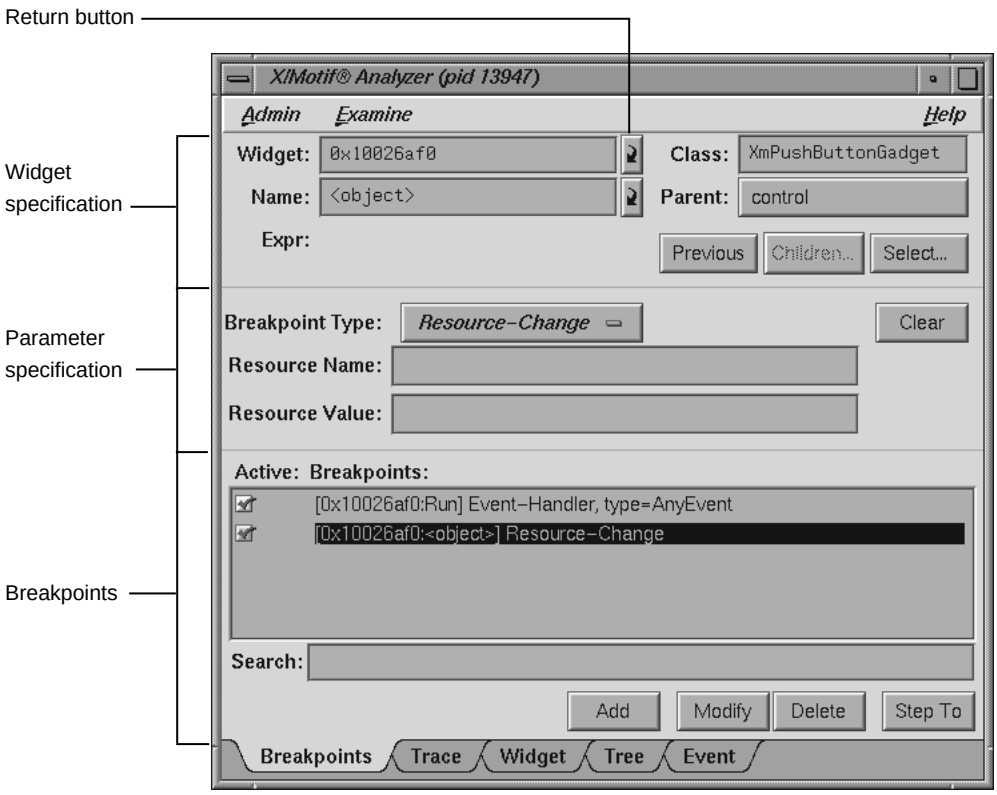

Figure 11-3 Adding a Breakpoint for a Widget

7. Click on the red breakpoint down arrow in the Annotation Column of the Main View window to remove the `Clock::timeout` breakpoint. If you click on the line, but not the down arrow, the breakpoint will be deleted; but the source pane will still display the arrow.
8. Click on the **Continue** button in the Main View window.
9. Click on the **Stop** button in the `Bounce` window.
10. Click on the **Run** button in the `Bounce` window. The process stops in the **Run** button's registered `activateCallback`. This is the routine that was passed to `XtAddCallback` routine. Notice that the **Callback** tab (for the Callback Examiner) is added to the tab list.

Setting Callback Breakpoints

1. Click on the **Breakpoints** list item (the `Active` box will be checked) to highlight the breakpoint in the **X/Motif Analyzer Breakpoint Examiner** window.
2. Delete the widget address in the **Widget** text field by backspacing over the text.
3. Click on the **Modify** button to change the `activateCallback` breakpoint to apply to all push-button gadgets `XmPushButtonGadget` (see in the **Class** text field) rather than just the **Run** button. See Figure 11-4.

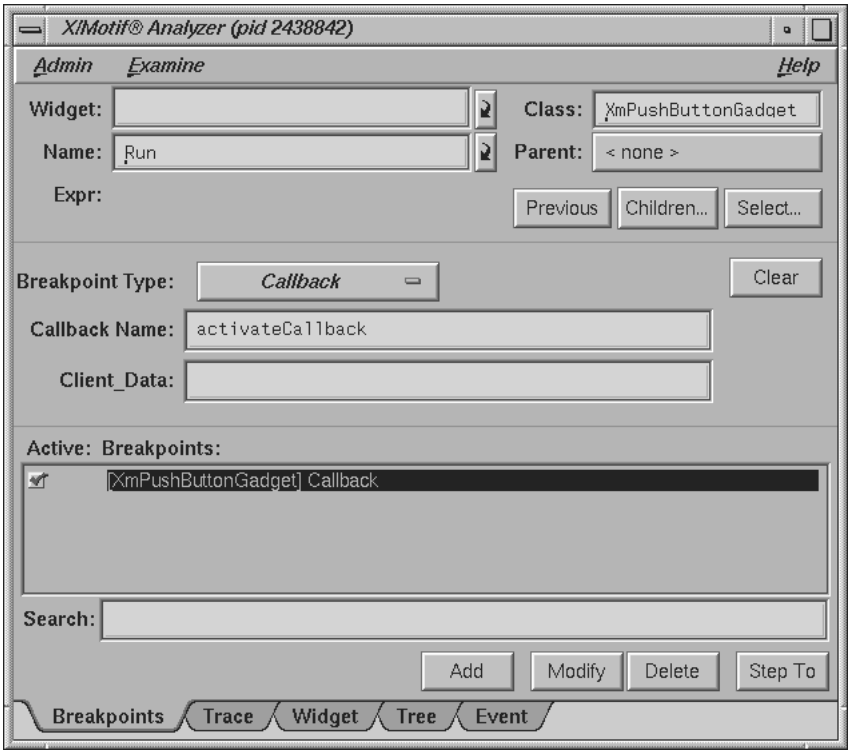

Figure 11-4 Setting Breakpoints for a Widget Class

4. Click **Continue** in the Main View window.
5. Click **Stop** in the `Bounce` window.

The process now stops in the **Stop** button's `activateCallback` routine.

6. Click the **Callback** tab in the **X/Motif Analyzer** window to go to the Callback Examiner. This examiner displays the callback context and the appropriate `call_data` structure (see Figure 11-5).

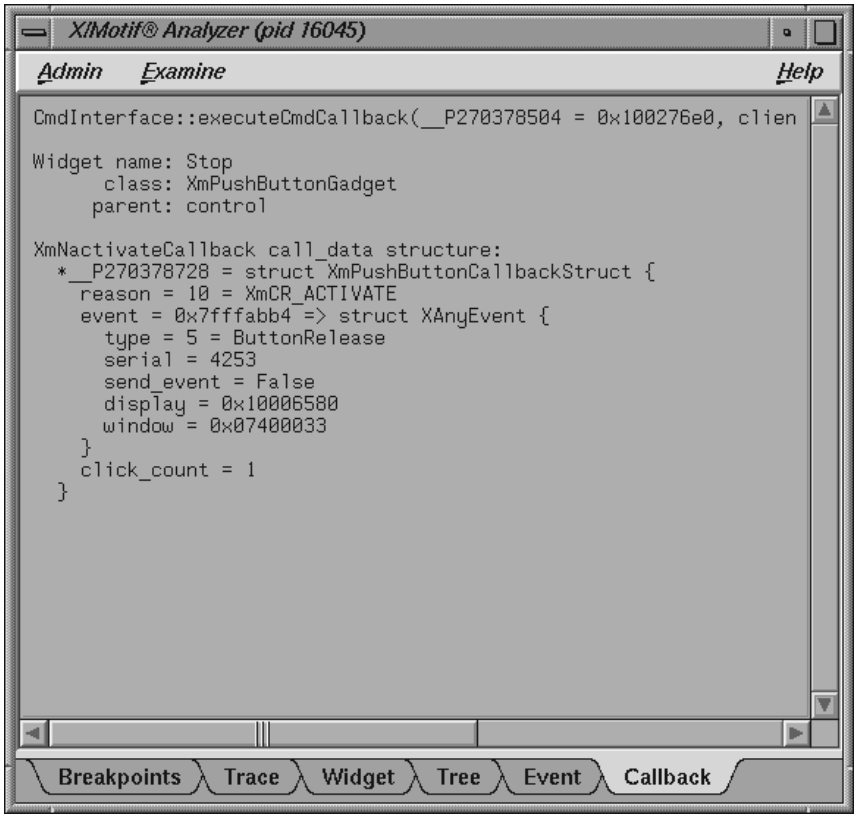

Figure 11-5 Callback Context Displayed by the Callback Examiner

7. Double-click the window value in the callback structure, fourth line from bottom.

8. Select **Examine > Window** in the Callback Examiner: . The X/Motif Analyzer displays the window attributes for that window (the window of the **Stop** button). Notice that the **Window** tab (for the Window Examiner) is added to the tab list. See Figure 11-6.

Note: You can also accomplish the same action by triple-clicking the window value in the callback structure of the Callback Examiner (Step 7). In general, triple-clicking on an address brings you to that object in the appropriate examiner.

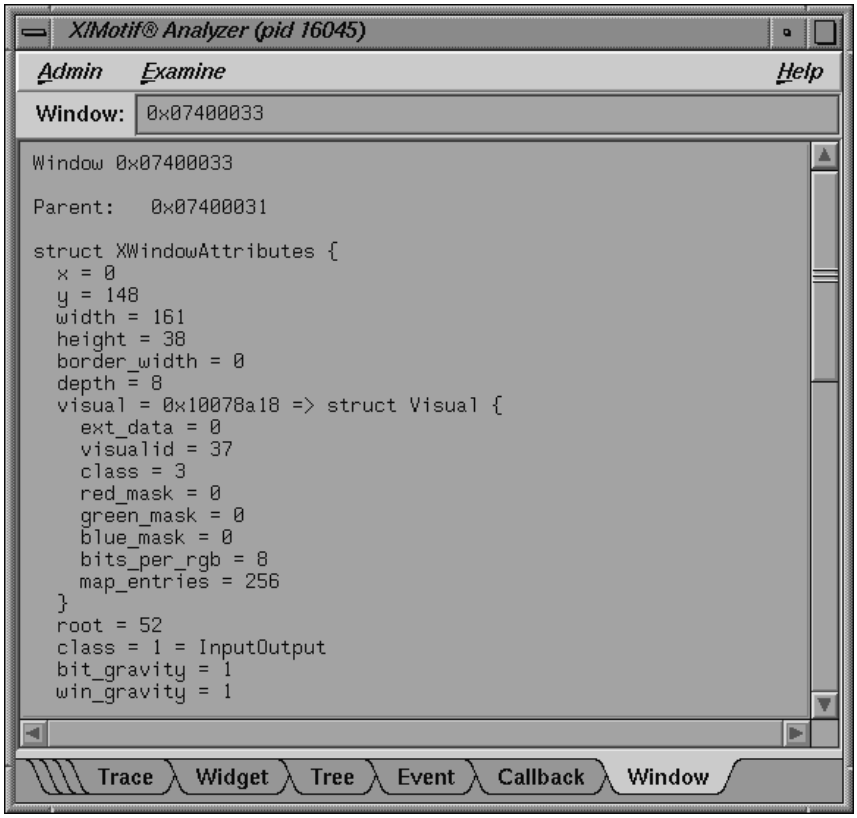

Figure 11-6 Window Attributes Displayed by the Window Examiner

Using Additional Features of the Analyzer

The following steps demonstrate additional Analyzer features.

1. In the **X/Motif Analyzer** window, click the **Widget** tab.

2. Double-click the `widget_class` value (on the fourth line) to highlight it.
3. Pull down **Examine > Widget Class**. The **X/Motif Analyzer** window displays the class record for the `XmPushButtonGadget` routine. Notice that the **Widget Class** tab (for the widget class examiner) is added to the tab list.

   The same action can be accomplished by triple-clicking the `widget_class` value in the Widget Examiner.
4. Triple-click the superclass value on the third line. The **X/Motif Analyzer** window displays the class record for `XmLabelGadget`, the superclass of `XmPushButtonGadget`.
5. Triple-click the superclass value on the right side of the third line. The **X/Motif Analyzer** window displays the class record for `XmGadget`, the superclass of `XmLabelGadget`.
6. Select the **Widget** tab to change to the Widget Examiner.
7. Triple-click the parent value on the fifth line. The **X/Motif Analyzer** window displays the `control` widget, the parent of **Run**. This action produces the same results as selecting the **control** text in the **Parent** text box.
8. Right-click on the tab overflow area (the area where the tabs overlap, to the far left of the tab list) as labeled in Figure 11-7, and select the **Breakpoints** tab.

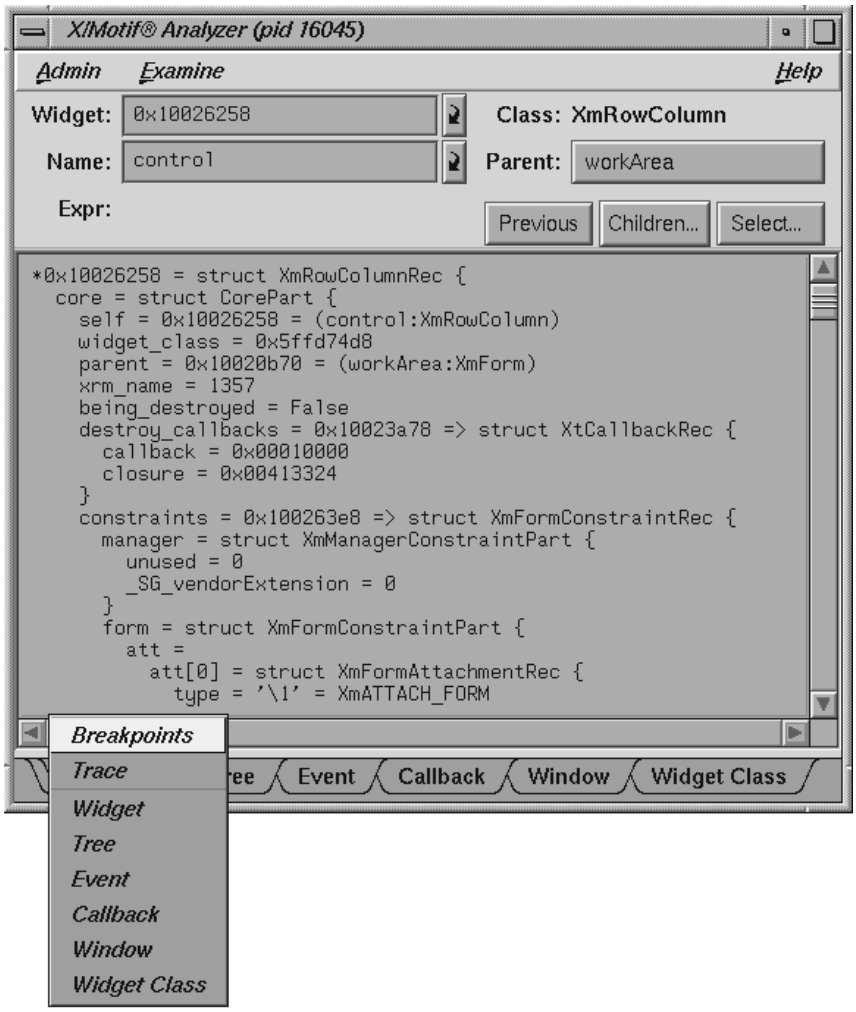

Figure 11-7 Selecting the Breakpoints Tab from the Overflow Area

9. Click on the word **Callback** in the **Breakpoint Type** text field of the **Breakpoints Examiner** window to bring up a submenu, and select **Resource-Change**.
10. In the **Class** text field, enter: **`Any`**.
11. In the **Resource Name** text field, enter: **`sensitive`**.

12. Click **Add**. This adds a breakpoint. The **Active: Breakpoints:** list should now include the following text:

    ```
    [Any] Resource-Change,name=sensitive
    ```

13. Click **Continue**. The status will update to **Stopped** in the `SetValues` routine, since the breakpoint set in the previous step was reached.

14. Select **Views > Call Stack** in the Main View window. Notice the call to `XtSetValues` on the second line (see Figure 11-8).

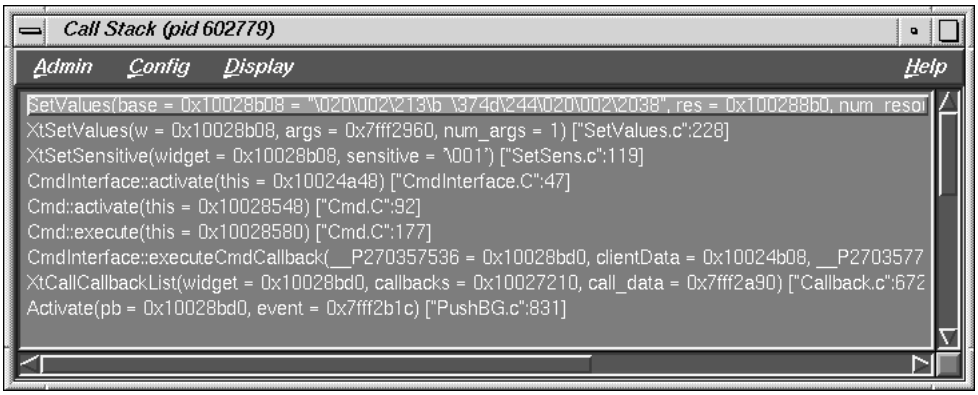

Figure 11-8 Breakpoint Results Displayed by the **Call Stack**

15. In the **Call Stack**, double-click the `Cmdinterface::activate` line (just below `XtSetSensitive`). This is where the sensitive resource was changed.

16. In the **Widget Examiner** window, double-click the widget address in the **Widget** text field, press backspace, enter **`_w`**, and press `Enter`. The X/Motif Analyzer displays the `Run` widget, which is the widget currently being changed.

17. Click **Continue** in the Main View window. The status will update to **Stopped** in the `SetValues` routine, since the breakpoint set in the previous step was reached again.

18. In the **Call Stack**, double-click on the `Cmdinterface::activate` line (just below `XtSetSensitive`).

19. Perform the following sub-steps:

a. Double-click in **Widget** text field of the **Widget Examiner**.

b. Press backspace.

c. Enter `_w`.

d. Press `Enter`.

The **X/Motif Analyzer** window displays the `Step` widget, which is the widget currently being changed.

Ending the Session

Select the following to close the X/Motif Analyzer: **Admin > Close**.

Select the following to exit the Debugger (from the Main View window): **Admin > Exit**. If you exit the Debugger first, you exit the X/Motif Analyzer as well.

For more information on the X/Motif Analyzer, see "X/Motif Analyzer Windows", page 214.

Customizing the Debugger

This section shows you how you may customize the WorkShop Debugger specifically to your environment needs.

Customizing the Debugger with Scripts

If there are Debugger commands or combinations of Debugger commands that you use frequently, you may find it convenient to create a script composed of Debugger commands. Debugger scripts are ASCII files containing one Debugger command and its arguments per line. A Debugger script can in turn call other Debugger scripts. There are three general methods for running scripts:

- Enter the `source` command and the filename at the Debugger command line. This is useful for scripts that you need only occasionally.
- Include the script in a startup file. This is useful for scripts that you want implemented every time you use the Debugger.
- Define a button in the graphical interface to run the script. Use this method for scripts you use frequently but apply only at specific times during a debugging session.

Using a Startup File

A startup file lets you preload your favorite buttons and aliases in a file that runs when the Debugger is invoked. It also is useful if you have traps that you set the same way each time. The suggested name for the startup file is `.cvdrc`. However, you can select a different name as long as you specify its path in the `CVDINIT` environment variable. The Debugger uses the following criteria when looking for a startup file:

- Checks the `CVDINIT` environment variable.
- Check for a `.cvdrc` file in the current directory.
- Checks for a `.cvdrc` file in the user's home directory.

Implementing User-Defined Buttons

You can implement buttons by providing a special Debugger startup file or by creating them on the fly within a debugging session. Buttons appear in the order of implementation in a row at the bottom of the control panel area. Currently, you can define only one row of custom buttons. Figure 12-1 is a typical example of the Main View window with user-defined buttons. The definitions for the user-defined buttons display in the Debugger command line area.

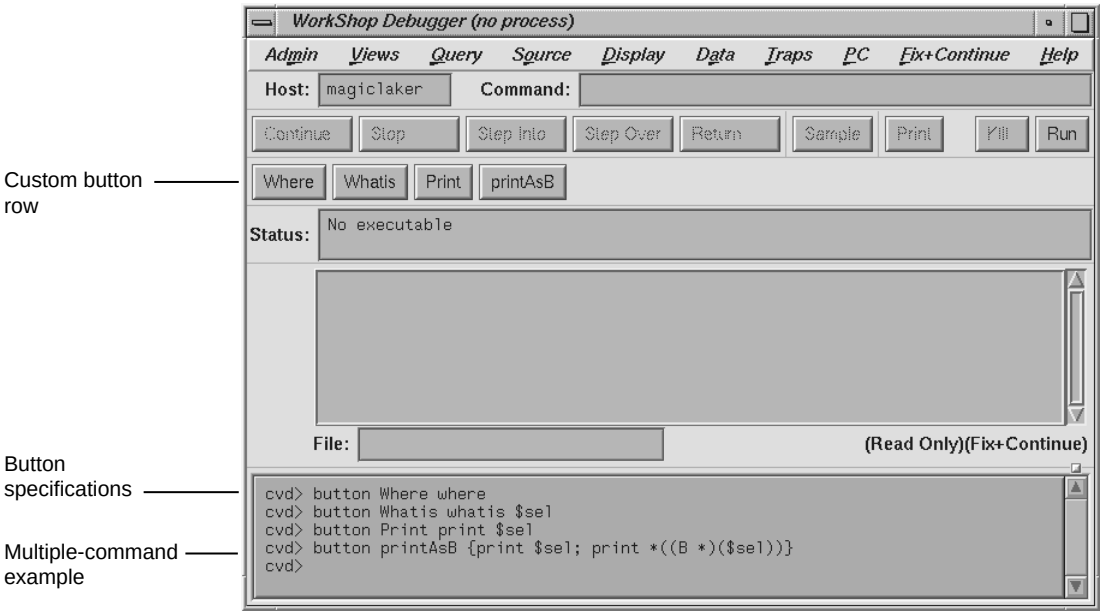

Figure 12-1 User-Defined Button Example

The syntax for creating a button is as follows:

```
button label command [$sel]
```

The syntax for creating a multiple-command button is as follows:

```
button label {command1 [$sel]; command2 [$sel]; ...}
```

The button command accepts the following options:

- *label*: specifies the button name. Button labels should be kept short since there is only room for a single row of buttons. There can be no spaces in a label.
- *command*: specifies one of the Debugger commands, which are entered at the command line at the bottom of the Main View window. See "Debugger Command Line", page 313.
- `$sel`: provides a substitute for the current cursor selection and should be appropriate as an argument to the selected command.
- *commandn...*: specifies Debugger commands to be applied in order. Commands must be separated by semicolons (;) and enclosed by braces ({}). The multiple-command button is a powerful feature; it lets you write a short script to be executed when you click the button.

The following command displays a list of all currently defined buttons:

```
% button
```

The following command deletes the button corresponding to the label:

```
% unbutton label
```

You might use this command if you needed room to create a new button. The effect of `unbutton` is temporary so that subsequently running the startup file reactivates the button.

The following command displays the definition of the specified button, if it exists. If the button does not exist, an error message is displayed:

```
% button label
```

Changing X Window System Resources

While there are many X Window System resources that you can change, we recommend that you avoid modifying these resources if at all possible. In some cases, there may be no way within WorkShop to make the desired change. If you must modify resources, the following X Window System resources for the Debugger and its views may be useful:

`*AllowPendingTraps:`

If set to true, enables support for pending traps.

The default value is false.

`*autoStringFormat`

If set to true, sets default format for `*char` results as strings in Expression View, the Variable Browser, and the Structure Browser.

The default format is the hexadecimal address.

`cvmain*sourceView*nameText.columns`

Sets the length of the **File** field in the Main View window.

The default value is 30 characters.

`Cvmain*disableLicenseWarnings` and `*disableLicenseWarnings`

Disables the license warning message that displays when you start the Debugger and the other tools.

`*editorCommand`

If you prefer to view source code in a text editor rather than in **Source View**, lets you specify a text editor.

The default value is the `vi` editor.

`*expressionView*maxNumOfExpr`

Lets you set the maximum number of expressions that can be read from a file by Expression View.

The default value is 25.

`*UseOldExprEval` and `*useOldExprEval`

When set to true, allows you to use the older, `dbx`–like, expression evaluator rather than the default C++ expression evaluator introduced in WorkShop version 2.6.4. Using the older evaluator may result in faster evaluation of some expressions.

The default value is false.

`*varBrowser*maxSymSize`

Lets you set the maximum number of variables that can be displayed by the Variable Browser.

The default value is 25.

The following resources apply to **Source View**:

`*svComponent*lineNumbersVisible`

Displays source line numbers by default.

`*sourceView*nameText.columns`

Sets the length of the **File** field in **Source View**.

The default value is 30 characters.

`*sourceView*textEdit.scrollHorizontal`

If set to true, displays a horizontal scroll bar in **Source View**.

`*tabWidth`

Sets the number of spaces for tabs in **Source View**.

The following resource applies to **Build View**:

`*buildCommand`

Is used to determine which program to used with `make`(1), `smake`(1), `clearmake`(1), and so forth.

The default value is `make`(1).

`*runBuild`

Specifies whether `cvmake`(1) begins its build immediately upon being launched.

The default value is true.

To change these resources, you need to set the desired value in your `.Xdefaults` file, and run the `xrdb`(1) command. Then, restart your application so that the resource gets picked up.

DUMPCORE Environment Variable

The DUMPCORE environment variable allows the Debugger to dump a core file in the event that there is a debugger execution problem during the debug session. To enable core files, enter either of the following before you startup your debug session:

- For the C shell, enter:

  ```
  % setenv DUMPCORE
  ```

- For the Korn shell or Bourne shell, enter:

  ```
  $ DUMPCORE=;export DUMPCORE
  ```

Debugger Reference

This chapter describes the function of each window, menu, and display in the Debugger's graphical user interface and describes the commands available on the Debugger command line (see "Debugger Command Line", page 313).

This chapter contains the following sections:

Main View Window

The major areas of the Main View window are shown in Figure A-1, page 172.

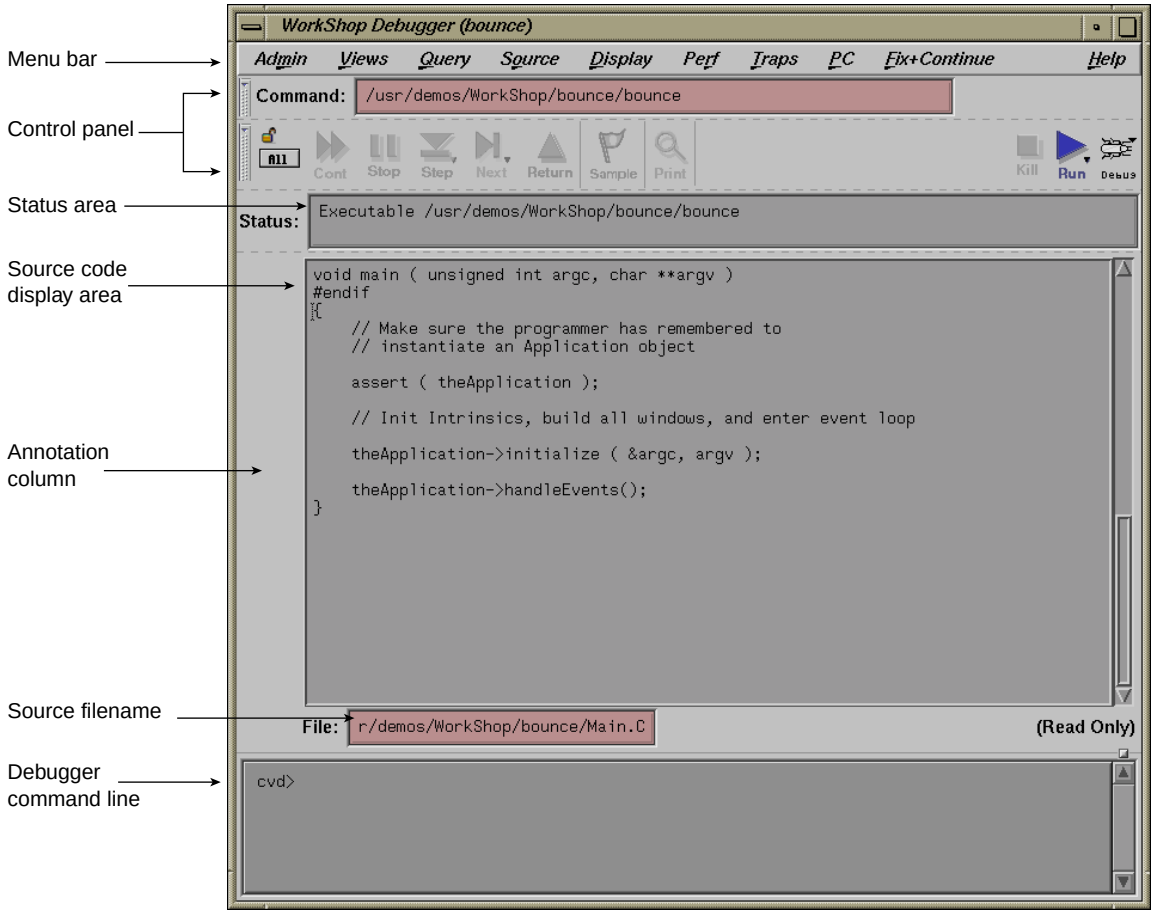

Figure A-1 Major Areas of the Main View Window

The Main View window contains a menu bar, from which you can perform a number of actions and launch windows. The menu bar contains the following menus, which are discussed in detail in later pages:

- "Admin Menu", page 179
- "Views Menu", page 181
- "Query Menu", page 183

The Main View window also contains several input fields, a source code display area, and buttons that trigger commonly used actions.

The Main View window contains the following items:

Command text field

Displays full pathname of the executable file that you are currently debugging, including any run-time arguments.

Debug button

Allows you to toggle among various modes. Right-click on the **Debug** button to display the **State Indicator menu** where the following mode choices are available:

- **Debug** runs the Debugger in Debug mode with no performance tools enabled.
- **Performance** mode causes performance data to be gathered and instrumented code to be generated for performance analysis while using the Debugger.
- **Purify** mode activates the Purify memory corruption analysis tool. The code displayed in the Main View window, **Source View** window, and so forth will be code generated by **Purify**. (This option appears only if Purify is installed on your system. Purify is not an SGI product nor is it part of the WorkShop package. It is a product of Rational Software and is neither available from nor supported by SGI.)

Stay Focused/Follow Interesting button

If the lock icon is locked, it indicates that the focus of Main View will attempt to stay focused on this thread. If the lock is unlocked, the debugger will follow the interesting thread. This means it will focus on threads that reach a user breakpoint.

All/Single button

If set to **All**, the **Cont**, **Stop**, **Step**, **Next**, and **Return** actions apply to all processor or threads. If set to **Single**, then only the currently focused process or thread will be acted upon. For more information about multiprocess and pthreaded programs, refer to Chapter 10, "Multiple Process Debugging", page 125.

Cont button

Continues execution of the current process or all processes in the program. When you click on the **Cont** button, the program runs either to a breakpoint exception or to termination. This button is active only after the running process(es) has stopped. If the program has not been run or has been killed, the **Cont** button is grayed out. If the target program has not yet started executing, use the **Run** button to start execution.

Stop button

Stops execution of the current running process(es). This button is valid only when a process(es) is running; otherwise the button is grayed out. Traps can also be set to stop the program at a specific location or on a particular condition.

Step button

Executes the code involving a single source line of the current process. If a function is encountered in the source line, or is the source line is a subroutine call, the process will step into a function or subroutine call and stop at the first executable line in the function or subroutine. The **Next** button can be used to continue to the next source line in the current file. If a trap is encountered while executing the step into command, the process is stopped where the trap was fired. The **Step** button is active only after the running process(es) has stopped; otherwise the button is grayed out.

When you right-click on the **Step** button, a menu pops up to allow you to choose the number of source lines to be stepped. If you choose the **N** menu entry, a dialog window is opened to allow you to enter a step value.

Next button

Executes the code involving a single source line of the current process. The current process continues to the next source line in the current file, and does not count any statements in functions that may be encountered in the source line. If the source line is a subrouting call, the process will stop at the next source line in the current file. The **Step** button can be used to step into a function or subroutine call and stop at the first executable line in the function or subroutine. If a trap is encountered while executing the step over command, the process is stopped where the trap was fired. The **Next** button is active only after the running process(es) has stopped; otherwise the button is grayed out.

When you right-click on the **Next** button, a menu pops up to allow you to choose the number of source lines to be stepped. If you choose the **N** menu entry, a dialog window is opened to allow you to enter a step value.

Return button

Continues execution of the process until the current function being executed returns. The process is stopped immediately upon returning to the calling function. All code within the current function is executed as usual. If a breakpoint is encountered, the action is canceled and the process is stopped where the breakpoint was fired. You can use this button only after the running process(es) has stopped; otherwise the button is grayed out. This action is not allowed if the executable is instrumented for performance analysis.

Note: For IRIX 6.5, this command button always returns all pthreads.

Sample button

Allows you to collect process state data to be used by the Performance Analyzer for program evaluation. You can use this button only when

the process(es) is running and the **Enable Data Collection** mode is set on the **Performance** panel; otherwise the button is grayed out.

See *ProDev Workshop: Performance Analyzer User's Guide* for more information.

Print button

Prints, in the **Source View** window, the value of any highlighted text, or the source pane of the Main View window.

Kill button

Kills the currently running process or all running processes in your debug session by sending the process(es) the equivalent of a `kill -9` signal. You can use this button if the process(es) is running or stopped; otherwise the button is grayed out.

Run button

Runs the program that you are currently debugging or all programs. After the initial run, allows you to rerun the program(s) while maintaining any breakpoints and command line arguments you have set.

Status area

Displays information about the process that you are debugging, such as process id, thread id, function name, list of arguments, location of the PC, and so forth,

Source Code area

Displays the source code for the program that you are currently debugging.

Annotation column

Clicking in this area displays information specific to a line number, such as breakpoints, location of the PC, and so forth.

File text field

Displays the name of the file shown in the source code area.

Command line area

Area of the Main View window where you can enter Debugger commands and view line-mode debugger message, and in which Debugger messages are displayed.

Show/Hide annotations button

This button (see Figure A-2, page 178) is visible only when you run or load a performance experiment (see the *ProDev Workshop: Performance Analyzer User's Guide* for more information on the performance tools).

This is a toggle button that shows or hides performance related annotations.

Figure A-2 Show/Hide Annotations Button in the Main View Window

Admin Menu

The **Admin** menu in the Main View window performs administrative and general management tasks dealing with processes, windows, and user preferences. The **Admin** menu provides the following selections:

Library Search Path

Controls where the Debugger looks for DSOs when you invoke the Debugger on an executable or core file. The **Library Search Path** dialog allows you to reset the `LD_LIBRARY_PATH` and `_RLD_ROOT` environment variables. You can also reset `_RLD_LIST` to control the set of DSOs that will be used by the program. See the `rld`(1) man page for more information on these variables. Any changes you make to these variables are propagated into **Execution View** when you run the program.

There are two ways to open this dialog. First, select **Admin > Library Search Path** from the Main View window menu bar, or the **Library Search Path** dialog opens automatically if you invoke the Debugger on an executable or core file and it is unable to find all of the required DSOs. In this case, an annotated list of required DSOs displays at the top of the dialog box with such status messages as `OK`, `Error: Cannot find library`, or `Error: Core file and library mismatch` (which indicates that the Debugger found a DSO that did not match the core file). Below this list are three to nine fields in which you can modify the value of the corresponding named environment variable.

Remap Path

Opens a dialog that allows you to enter a new pathname.

Multiprocess View

Displays the **Multiprocess View** window, which allows you to control processes and threads. You should note that if you exit from **Multiprocess View**, you will exit from your debugging session. More additional information about **Multiprocess View**, refer to "**Multiprocess View**", page 197.

GLdebug

Provides a toggle to turn on `GLdebug`. `GLdebug` is a graphical software tool for debugging application programs that use the IRIS Graphics Library (GL). `GLdebug` locates programming errors in executables when GL calls are used incorrectly.

Attach/Switch Process

Changes the current process or attaches to a process. You will be queried for the new process ID. You can select one from the list of items presented, type one in or paste one in from another window. Switching processes changes the session.

Load/Switch to Executable

Changes the current executable or loads an executable. This option also allows you to debug a different core file.

Detach

Releases the process from the Debugger. This allows you to make changes to the source code. You must detach the process before you recompile the program.

Load Settings

Allows you to use the previously saved preference settings from a file you choose in the **Load Settings** dialog.

Save Settings

Allows you to save the current preference settings to an initialization file used when the Debugger is first started, or any file you choose through this dialog. These can include such items as window sizes, current views, window configurations, and so on.

Iconify

Iconifies all session views.

Raise

Brings all session view windows to the foreground and displays any iconified windows.

Launch Tool

Allows you to run other WorkShop tools. You can switch to the other tools by selecting **Build Analyzer**, **Static Analyzer**, **Performance Analyzer**, or **Tester**. Selecting **Debugger** allows you to start another debugging session. If you have ProDev ProMP (formerly called WorkShop Pro MPF) installed on your system, the **Parallel Analyzer** selection is also available.

Close

Closes the Main View window.

Exit

Closes all views in the session and terminates the session.

Views Menu

The **Views** menu in Main View window provides the following selections for viewing the process(es) and their corresponding data:

Array Browser

Displays values from an array or array-slice in a two-dimensional spreadsheet and optionally in a three-dimensional representation; that is, a bar graph, surface, multiple lines, or points in space. These help you pick out bad data more readily. Arrays can contain up to 100 x 100 elements.

Call Stack

Displays the call stack along with parameters to the calls. If you double-click an entry in the stack, you switch the current context to that entry and you can check the state of variables.

Disassembly View

Displays assembly code corresponding to the source code.

Exception View

Displays an Ada-specific window used for exception handling.

Execution View

Displays the iconified **Execution View** window, which handles the input and output of the target process.

Expression View

Evaluates expressions in Fortran, C, or C++. To enter an expression, select it in the source code display and paste it into the **Expression View** field, using the middle mouse button.

File Browser

Displays a list of source files and library routines used by the current executable. Double-click a source file in the list to load it directly into the source display area in Main View or **Source View** windows. The **Search field** allows you to find files in the list quickly.

Memory View

Displays the value at a given memory address.

Process Meter

Monitors the resource usage of a running process without saving the data. (Used with the Performance Analyzer.)

Register View

Displays the values stored in the hardware registers for the target process.

Signal Panel

Displays the signals that can occur. You can specify which signals trigger traps and which are to be ignored.

Source View

Displays source code. Allows you to set traps, perform searches, and inspect source code without losing information in the Main View window.

Structure Browser

Displays data structures in a graphical format. You can de-reference pointers by double-clicking.

Syscall Panel

Allows you to set traps at the entry to or exit from system calls.

Task View

Brings up an Ada-specific view that provides task and callstack information for processes.

Trap Manager

Allows you to set, edit, and manage traps. The Trap Manager is used by both the Debugger and the Performance Analyzer.

Variable Browser

Displays values of local variables and parameters for the current context.

X/Motif Analyzer

Provides you with specific debugging support for X/Motif applications. There are various examiners for different X/Motif objects, such as widgets and X graphics contexts, that might be difficult or impossible to inspect using ordinary Debugger functionality.

Query Menu

The **Query** menu allows you to perform some of the queries available in the Static Analyzer. These queries are convenient if you have previously built a `cvstatic` fileset. However, if you need to build the fileset from scratch, the process becomes more involved. For complete information about using the Static Analyzer, see the *ProDev WorkShop: Static Analyzer User's Guide* and the `cvstatic`(1) man page.

With a current fileset, you can double-click any defined entity in the source code, select the **Where Defined?** option from the submenu appropriate to its type, and the source code display area will scroll to the location where the item is defined.

Source Menu

The **Source** menu in the Main View window provides the following selections to manage source code files:

Open

Loads a source file.

Open Recent

Provides you with a popup dialog that gives you a selection of recently-opened files from which to choose.

Save

Records changes made during the debugging session to the source file. You must first select **Make Editable**, which appears in the **Source** menu when the file is read-only.

Save As

Records changes made during the debugging session to the source file under a different filename. You must first select **Make Editable**, which appears in the **Source** menu when the file is read-only.

Save As Text

Records the information in the display area as a text file.

Insert Source

Inserts the text of a file within your current file. You must first select **Make Editable**, which appears in the **Source** menu when the file is read-only. You must first select **Make Editable**, which appears in the **Source** menu when the file is read-only.

Fork Editor

Starts your default editor on the current file. The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d`, which means run `vi` in a `wsh` window and scroll to the current line. If the editor allows you to specify a starting line, enter `%d` in the resource to indicate the new line number.

Recompile

Displays the **Build View** window, which allows you to compile the source code associated with the current executable.

Make Read Only / Make Editable

Toggles the source code displayed between read-only and writable states so that you can edit your code.

Search

Searches for a literal case-sensitive, literal case-insensitive, or regular expression. After you have set your target and clicked **Apply** (or pressed `Enter`), each instance is marked by a search target indicator in the scroll bar. You can search forward or backward in the file by clicking the **Next** and **Prev** buttons. You can also click an indicator with the middle mouse button to scroll to that point. Clicking **Reset** removes the search target indicators.

Go To Line

Allows you to scroll to a position in the source code by specifying a line number. **Go To Line** brings up a dialog box.

You can enter a line number or use the slider at the top of the box to select a line number. You do not have to display line numbers to use this feature.

Versioning

Provides access to the configuration management tool, if you have designated one.

Type the following at the **Execution View** prompt:

```
/usr/sbin/cvconfig [rcs | sccs | ptools | clearcase]
```

Note: You must have `root` permissions to run `cvconfig`.

The **Versioning** submenu appears.

Selecting any submenu option displays a shell in which you can access the configuration management tool. The following selections are available on the submenu:

- **CheckIn** — Saves the source file and checks it into the database as a new version.

- **CheckOut** — Recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.
- **UncheckOut** — Cancels the checkout, with no changes registered.

Display Menu

The **Display** menu in the Main View window provides the following selections to annotate the displayed source code:

Show Line Numbers/Hide Line Numbers

Displays or hides line numbers in the annotation column corresponding to the source code.

Show Toolbar

Allows you to choose the format type for the toolbar. The options are **Text Only**, **Icons Only**, or **Icons and Text**. **Text Only** was the only format for the toolbar prior to release 2.9. **Icons Only** and **Icons and Text** are options which display icon options for more visual debugging.

Show Tooltips/Hide Tooltips

This menu item enables or disables the context sensitive pop-up help option. Some of the key menu items, buttons, and data entry areas have pop-up help statements attached to them to give the user hints on what to use them for or how to use them.

Preferences

Displays the **Annotations Preferences** dialog box, which allows you to show or hide column annotations and menus specific to the different WorkShop tools. If you have purchased ProDev ProMP, you can display and manipulate loop indicators. The **Performance** toggle displays experiment statistics. The Tester module allows you to see coverage statistics. Turning off the **Performance** toggle deletes the performance annotations from the **Source View**.

Hide Icons/Show Icons

Hides or displays the annotation column, which is located to the left of the source code display area.

Perf Menu

The **Perf** (Performance) menu (see Figure A-3, page 188) includes the following menu selections:

Select Task submenu

Allows you to choose the task for your performance analysis. The choices available are shown in Figure A-3, page 188. You may select only one task per performance analysis run. If none of the given tasks satisfy your requirements, you can choose **Custom**, which brings up the configuration dialog open to the **General** tab. From here, you can design your own task requirements.

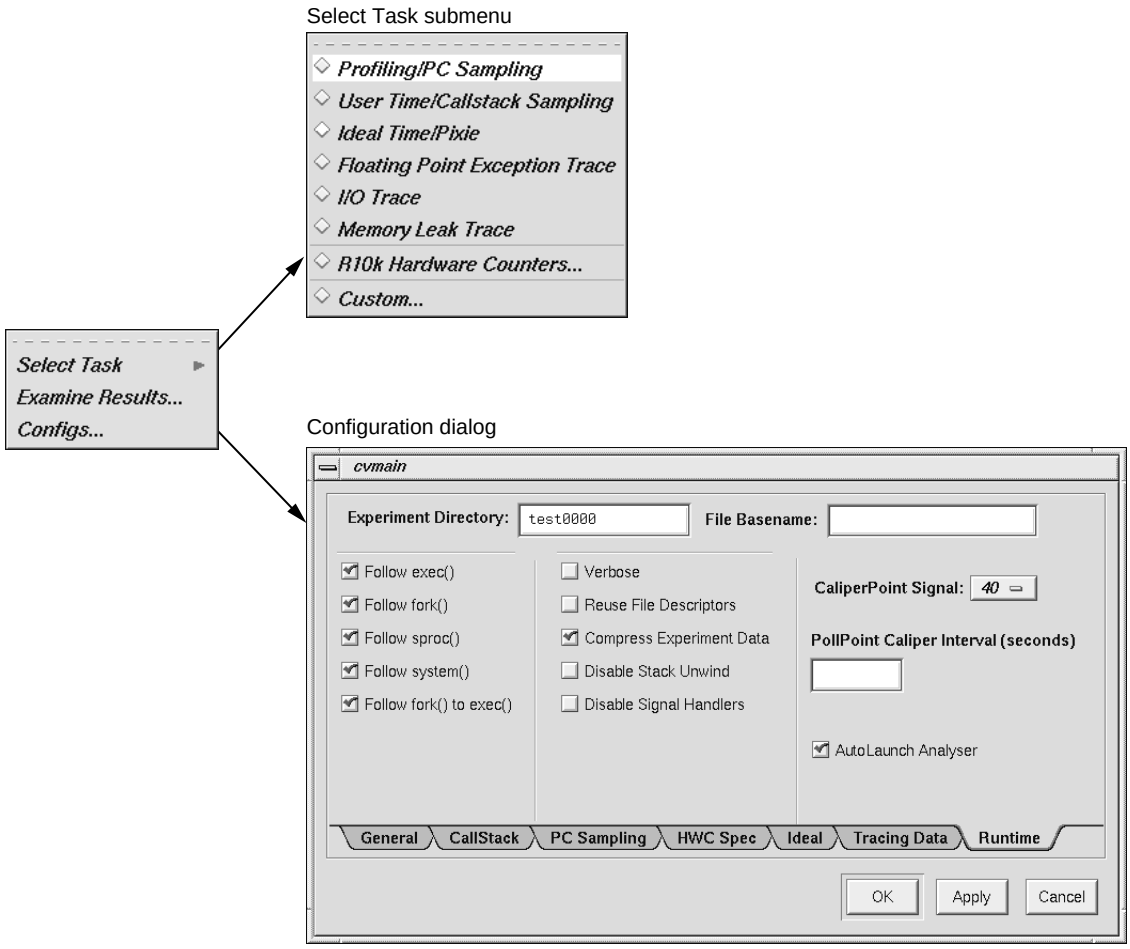

Figure A-3 Perf Menu and Subwindows

Examine Results

Launches the Performance Analyzer. For complete information about the Performance Analyzer, see the *ProDev Workshop: Performance Analyzer User's Guide*.

Configs

Brings up the configuration dialog open to the **Runtime** tab as shown in Figure A-3, page 188. The dialog opens with the **Experiment Directory** text field filled in with a default value. The Performance Analyzer provides a default directory named `test0000`. If you use the default or any other name that ends in four digits, the four digits are used as a counter and will be incremented automatically for each subsequent experiment.

Traps Menu

The **Traps** menu (see Figure 5-1, page 69) offers the **Set Trap** and **Clear Trap** submenus and the **Group Trap Default** and **Stop All Default** menu options.

The **Set Trap** submenu offers submenus for managing breakpoints and sample points. The following submenu selections are available:

Stop

Sets a breakpoint at a designated line in your source code. To set a breakpoint at a line displayed in the Main View or **Source View** windows:

1. position the cursor on the appropriate line in the source code display area
2. select the **Set Trap** submenu
3. choose the **Stop** option

The preferred method for setting a breakpoint is to click in the annotations area of the Main View window, across from the line at which you want to set the breakpoint.

Stop At Function Entry

Sets a breakpoint at the beginning of a function. To set a breakpoint at a function, click on the function name in the source code display area and select the **Set Trap** submenu, then choose the **Stop At Function Entry** option.

Stop At Function Exit

Sets a breakpoint at the end of a function. To set a breakpoint at a function exit, click on the function name in the source code display area and select the **Set Trap** submenu, then choose the **Stop At Function Exit** option.

Sample

Sets a sample trap at a line displayed in the Main View or **Source View** windows. To set a sample trap:

1. highlight on the appropriate line
2. pull down the **Set Trap** submenu
3. select the **Sample** option

Sample At Function Entry

Sets a sample trap at the beginning of a function. To set the sample trap, highlight the function name in the source code display area, then pull down the **Set Trap** submenu and select the **Sample At Function Entry** option.

Sample At Function Exit

Sets a sample trap at the end of a function. To set the sample trap, highlight the function name in the source code display area, then pull down the **Set Trap** submenu and select the **Sample At Function Exit** option.

The **Clear Trap** submenu contains selections that allow you to delete a trap on the line containing the cursor. You must designate **Stop** or **Sample** trap type, since both types can exist at the same location, appearing superimposed on each other. The following submenu selections are available:

Stop

Designates the stop trap type.

Sample

Designates the sample trap type.

The last two menu options allow you to specify the following items:

Group Trap Default

Interacts with **Source View**. If set to true, all subsequent **Source View** trap requests will be group traps. That is, all members of the process group will apply this trap. This option is the same as typing `stop pgrp in` *filename* from the command line. Default is false unless you are using IRIX 6.5 pthreads, when the implied setting is always true.

Stop All Default

Interacts with **Source View**. If set to true, all subsequent **Source View** trap requests will apply the `Stop All` command to the trap. That is, whenever this trap is encountered, all other members of the process group also will be stopped. This option is the same as typing `stop all in` *filename* from the command line. Default is false unless you are using IRIX 6.5 pthreads, when the implied setting is always true.

If both of the default options are set to true, it is the same as typing `stop all pgrp in` *filename* from the command line.

PC Menu

The **PC** (program counter) menu in the Main View window provides the following selections for controlling the execution of a process:

Continue To

Continues the process to the selected point in the program unless some other event interrupts. Select a line by clicking on it. The process must be stopped before you can use **Continue To**.

Jump To

Goes directly to a selected point within the same function, jumping over intervening code. Then the Debugger waits for a command to resume execution. Select a line by clicking on it.

Fix+Continue Menu

The **Fix+Continue** menu offers the following menu selections:

Edit

Allows you to edit text using the Debugger editor.

External Edit

Allows you to edit text by using an external editor. The default editor is `vi`, but can be changed by using the **Set Edit Tool** pop-up menu in the **Admin** menu of the **Status** window. See "Fix+Continue Status Window", page 302, for further information.

Parse and Load

Compiles your modified program and loads it for execution. You can execute the modified program by clicking on the **Run** or **Continue** buttons in the Main View window.

Show Difference submenu

Allows you to see the difference between the original code and your modifications. See "**Show Difference** Submenu", page 193, for further information.

Edited<–>Compiled

Enables or disables your changes. This switch allows you to see how your application executed before and after the changes you made.

Save As

Allows you to save your changes to a file. You can save changes to the current source file (the default) or to a separate file.

Save All Files

Launches the **Save File+Fixes As** dialog that allows you to update the current session and save all the modifications.

View submenu

Allows you to change to different views. Fix+Continue supports status, message, and build environment windows. See "View Submenu", page 193, for further information.

Preferences submenu

Allows you to set your Fix+Continue preferences. See "Preferences Submenu", page 194, for further information.

Cancel Edit

Takes you out of edit mode and cancels any changes you have made.

Delete Edits

Deletes any modifications that you made.

Show Difference Submenu

This submenu allows you to view differences between your original and your modified code. It contains the following options:

For Function

Opens a window that shows you the differences between the original source and your modified source.

For File

Opens a window that shows you the differences between the original source file and your modified version.

Set Diff Tool

Launches the **Fix+Continue Preferences Dialog** that allows you to set the tool that displays code differences. The default is `xdiff`(1). For further information on the **Fix+Continue Preferences Dialog**, see "Preferences Submenu", page 194.

View Submenu

This submenu allows you to open different Fix+Continue view windows. It contains the following options:

Status Window

Launches the **Fix+Continue Status** window. See "Fix+Continue Status Window", page 302, for more information.

Message Window

Launches the **Fix+Continue Message** window. See "Fix+Continue Error Messages Window", page 307, for more information.

Build Environment Window

Launches the **Fix+Continue Build Environment** window. See "Fix+Continue Build Environment Window", page 308, for more information.

Preferences Submenu

This submenu allows you to set various options for the Fix+Continue environment, such as the difference tool, the external editor command, and so on. The menu contains the following options:

Show Preferences

Launches the **Fix+Continue Preference Dialog** that displays preferences currently enabled for the session, and allows you to change the settings. The following preferences are available through the dialog:

- **External Editor Command** text field that allows you to choose your text editor. The default is `vi`.
- **File Difference Tool** text field allows you to choose the tool to use when comparing code. The default is `xdiff`(blank).
- **Copy Traps On Previous Definition** toggle allows you to edit and parse code. When Fix+Continue copies traps from the old definition to the new one by mapping old lines to new lines. (This mapping is the same as what can be generated using the UNIX `diff` utility.) If **Copy Traps On Previous Definition** is on and the mapped line the new definition is modified, then Fix+Continue will look at the switch.
- **Copy Traps Even On Changed Lines** toggle causes the debugger to copy traps onto a mapped line.
- **Continue Even If Line Has Changed** toggle allows you to edit and compile code in which your program is currently stopped. Fix+Continue can continue in the new definition provided some conditions are satisfied. The line from which the program continues depending on the mapping from the line in which it stopped. In case it can continue in the new definition from a line which you have modified, Fix+Continue consults this toggle to

determine whether to continue in the new or old definition. This toggle allows you to override the default behavior.

- **Warn Unfinished Edits Before Run** toggle pops up a warning dialog before a run if you have unfinished edits.
- **Warn Unfinished Edits Before Continue** toggle pops up a warning dialog before a continue if you have unfinished edits.
- **Save deactivated code during File Save** toggle save old code. The Fix+Continue file save substitutes new definitions in place of old ones. If you want to save your original functions in the same file, this switch allows you to save the old (original or compiled) code under an `#ifdef`. When you compile, the old code will not get compiled. You can manually edit the source to use the old definition in any way you desire.

Reset To Factory Defaults

Sets preferences to the installed defaults.

Save Preferences

Brings up the **File** dialog that allows you to save your preferences to a file.

Load Preferences

Brings up the **File** dialog that allows you to load preferences from a file.

Keyboard Accelerators

Use the accelerators in to issue Fix+Continue commands directly from the keyboard. The accelerators are listed alphabetically by command.

Table A-1 Fix+Continue Keyboard Accelerators

Command	Key Sequence
Cancel Edit	`Alt+Ctrl+q`
Edit	`Alt+Ctrl+e`
Parse And Load	`Alt+Ctrl+x`

Help Menu

The **Help** menu provides the following options:

Click for Help

Provides information on the selected window or menu.

Overview

Provides general information on the current tool.

Index

Displays the entire list of help topics, alphabetically, hierarchically, or graphically.

Keys & Shortcuts

Lists the keys and shortcuts for the current tool.

Product Information

Provides copyright and version number information on the tool.

Some Additional Views

This section discusses some of the additional views available through the Debugger: the **Execution View**, **Multiprocess View**, **Source View**, and **Process Meter**.

Execution View

The **Execution View** window is a simple shell that allows you to set environment variables and inspect error messages. If your program is designed to be interactive using standard I/O, this interaction will take place in the **Execution View** window. Any standard I/O that is not redirected by your Target command is displayed in the **Execution View** window. **Execution View** is launched (and iconified) automatically with the Debugger.

Multiprocess View

WorkShop supports debugging of multiprocess applications, including pthreaded programs and processes spawned with either `fork` or `sproc` commands.

Multiprocess debugging is supported primarily through the **Multiprocess View** window. To display this window, select **Multiprocess View** from the **Admin** menu of the Main View window. **Multiprocess View** displays a hierarchical view of your pthreaded application. Pthreaded processes are marked with a folder icon. Clicking the folder changes the view to show that process's pthreads. Clicking on a thread opens a call stack for that thread.

For each process or thread, clicking the right mouse button brings up a menu that applies to the selected item. This menu is a duplicate of the **Process** menu. See "Controlling Multiple Processes", page 206 for more information.

Figure A-4 Process Menu

Status of Processes

When the **Multiprocess View** window comes up, it lists the status of all processes in the process group. This view includes the following information:

PID	Shows the process identifier (PID).
PPID	Lists the parent process PIDs.
Status/State	Shows **Status** or **State** depending on how you have set your preferences on the Multiprocess View **Preferences** menu, brought up by selecting **Preferences** from the **Config** menu. **Status** is user–level status and **State** is the kernel–level status.
Name	Identifies the process by file name.
Function/PC	Indicates the current function and program counter (PC) for any stopped processes.

The following **Status** and **State** conditions are possible:

Status	**State**
Running	**RUNNING**
Stopped	**RUNNING**
Stopped on breakpoint	**RUNNING** (but at a trap pc)
Waiting to terminate	**JOIN**
Thread terminated	**DEAD**
Waiting on kernel	**READY**
Waiting on mutex	**MUTEX-WAIT**
Sleeping in system call	**RUNNING**

Multiprocess View Control Buttons

The **Multiprocess View** window uses the same control buttons as are in the Main View window with following exceptions:

- There are no Run, Return, or Print buttons in the **Multiprocess View**.
- The buttons in this view apply to all processes as a group.

- Using a control button in the **Multiprocess View** window has the same effect as clicking the button in each process's Main View window.

These buttons operate identically to those described for the Main View window, "Main View Window", page 171, with the **All** option effect. Refer to that section for descriptions of these buttons.

Multiprocess View Administrative Functions

The Admin menu in the **Multiprocess View** window lets you perform several administrative functions. Only the **Save as Text**, **Close**, and **Exit** items are described here. All other options perform as those found in the Admin menu of the Main View window, described in "Admin Menu", page 179.

Save as Text

The process status list from in the **Multiprocess View** window is saved to the file you select using the Save Text dialog.

Close

Closes the **Multiprocess View** window only.

Exit

Exits all views in the session and terminates the session.

Controlling Preferences

The **Preferences** option in the **Config** menu brings up the **Multiprocess View Preferences** dialog that allows you to control when processes are added to the group and specifies their behavior. This option also contains a **Save** option that allows you to save your preferences.

The **Multiprocess View** preference options are:

Stack Depth

Allows you to set how many lines of the call stack should be displayed when opening the call stack. Default is 10.

Levels to open

Allows you to specify the number of levels of hierarchy to be displayed. For pthreaded programs, you can display information by

program, threads, and callstack. For non-pthreaded programs, you can display information by process and callstack. Default is one level.

Attach to forked processes

Automatically attaches new processes spawned by the `fork` command to the group. (Note that processes spawned by `sproc` are always attached.) Default is off.

Copy traps to forked processes

Copies traps you have set in the parent process to new `forked` processes automatically. Alternatively, if you create parent traps with **Trap Manager** and specify `pgrp`, then the children inherit these traps automatically, regardless of the state of this flag. Default is off.

Copy traps to sproc'd processes

Copies traps you have set in the parent process to new `sproc'd` processes automatically. As in the previous option, if you create parent traps with the **Trap Manager** and specify `pgrp`, the children inherit these traps automatically, whether this flag is set or not. Default is on.

Resume parent after fork

Restarts the parent process automatically when a child is `forked`. Default is on.

Resume child after attach on fork

Restarts the new `forked` process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached. Default is on.

Resume parent after sproc

Restarts the parent process automatically when a child is `sproc'd`. Default is on.

Resume child after attach on sproc

Restarts the new `sproc'd` process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached. Default is on.

Combine threads at same location

Applies a collapsing algorithm to display threads stopped at the same location at the same time. (It is possible for threads to arrive at the same location through different logical routes.) Default is on.

Show Thread Status vs Thread State

Displays thread status, which is the user–level status as opposed to showing thread state, which is the kernel–level status. Default is status.

Show/Hide Buttons

When this option is **ON**, the **Continue All**, **Stop All**, **Step Into All**, **Step Over All**, **Sample All**, and **Kill All** buttons appear near the top of the **Multiprocess View** window.

Show/Hide Header Information

When this option is **ON**, column headings display above the list of processes/threads.

Source View

The **Source View** window is brought by choosing **Views > Source View** from the Main View window menu bar. By default, a copy of the source on display in the Main View window source pane is displayed in this window.

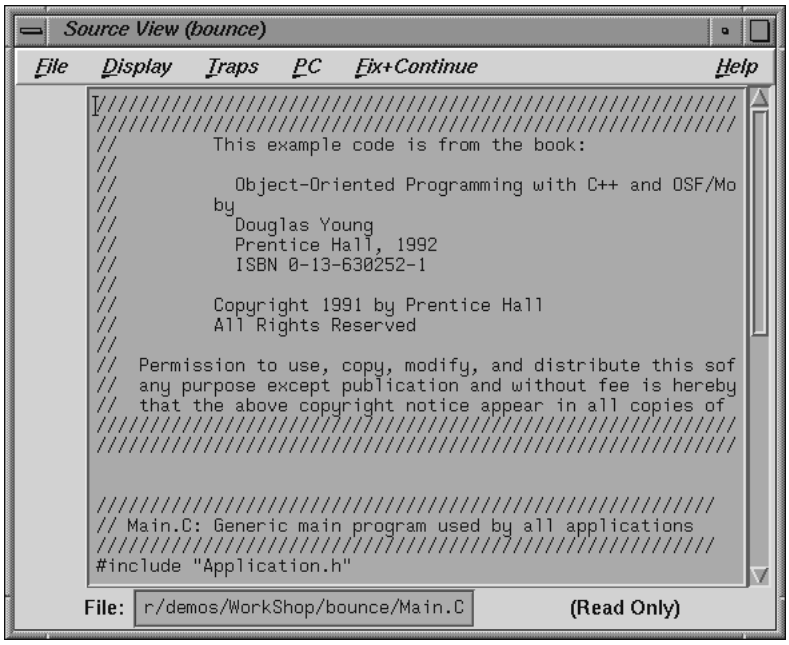

Figure A-5 Source View Window

The **Source View** menu bar contains selections duplicated from the Main View window: **Display**, **Traps**, **PC**, and **Fix+Continue**. Each of these menus has the same functionality as its counterpart in the Main View window (see "Main View", page 310). The only new menu selection is the **File** menu described below:

Open

Launches the `Open` dialog that allows you to choose a file to load into **Source View**.

Save

Records changes made to the file during the current debugging session. You must first select **Make Editable** from this **File** menu when the file is read only.

Save As

Records changes made during the debugging session to the source file under a different file name, the name of which you can enter or select using the **Save As Text** dialog.

Save As Text

Records information in the display area as a text file, the name of which you can enter or select using the **Save As Text** dialog.

Open Separate

Launches the **Open Separate** dialog that allows you to create a new **Source View** with the contents of a different source file.

Insert File

Inserts the text of a file within your current file. This item description is available only if the file is editable. The **Make Editable** item description from this **File** menu can be used to switch the file from read only.

Clone

Clones the current window.

Fork Editor

Starts your default editor on the current file. The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d`, which means run `vi` in a `wsh` window and scroll to the current line. If the editor allows you to specify a starting line, enter **`%d`** in the resource to indicate the new line number.

Recompile

Displays the **Build View** window that allows you to compile the source code associated with the current executable.

Make Editable

Toggles the source code displayed between **(Read Only)** and **(Editable)** so that you can edit your code.

Search

Searches for a literal case-sensitive, literal case-insensitive, or regular expression. After you have set your target and clicked **Apply** (or pressed `Enter`), each instance is marked by a search target indicator in the scroll bar. You can search forward or backward in the file by clicking the **Next** or the **Prev** button. You can also click an indicator with the middle mouse button to scroll to that point. Clicking **Reset** removes the search target indicators.

Go To Line

Launches the **Go To Line** dialog that allows you to go to a specific line in the source. You can type in the line, or select the line number via the slider bar.

Versioning

Provides access to the configuration management tool, if you have designated one.

The `cvconfig` script allows you to designate ClearCase, RCS or SCCS. Type the following:

```
/usr/sbin/cvconfig [rcs | sccs | ptools |  clearcase]
```

You must have `root` permissions to run `cvconfig`.

Selecting any option displays a shell in which you can access the configuration management tool. The selections in the submenu are:

Versioning submenu

Contains the following options:

- **CheckIn** — Saves the source file and checks it into the database as a new version.
- **CheckOut** — Recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.
- **UncheckOut** — Cancels the checkout, with no changes registered.

Close

Dismisses the **Source View** window.

Process Meter

The **Process Meter** window is brought up by choosing **Views > Process Meter** from the Main View menu bar. The **Process Meter** monitors resource usage of a running process without saving the data. Figure A-6, page 205, shows the Process Meter in its default configuration (with only the **User Time** and **Sys Time** fields active).

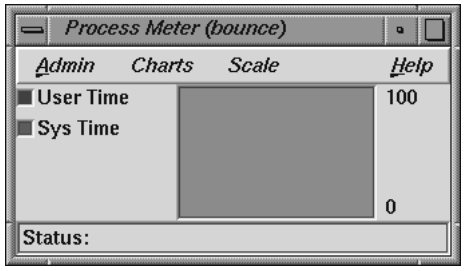

Figure A-6 Process Meter

The **Process Meter** contains its own menu bar that contains the **Admin**, **Charts**, **Scale**, and **Help** menus. The **Admin** menu is the same as that described in "Admin Menu", page 208. The **Help** menu is the same as that described in "Help Menu", page 196. The other menus are described in the following sections.

Charts Menu

The **Charts** menu contains a set of toggles that allow you to choose which charts are displayed in the **Process Meter** window. You can display as many charts simultaneously as you wish. The following choices are available:

User/Sys Time (the default)
Major/Minor Faults
Context Switches
Bytes Read/Written
Read/Write Sys Calls
Other Sys Calls

Total Sys Calls
Signals
Process Size

Scale Menu

The **Scale** menu allows you to set the time scale for the processes displayed in the **Process Meter** window. A menu allows you to choose a time scale from 2 seconds to 10 minutes.

Controlling Multiple Processes

The **Process** menu allows you to control processes and threads.

Figure A-7 Process Menu

The **Process** menu has the following options:

Change focus to this entry

Opens a dialog that allows you to switch the process or thread currently focused on in the Main View window to the process or thread selected in the **Multiprocess View** window. Selecting a call stack entry changes the Main View window's focus to that process or thread and positions the **cvmain** window at the offset of the selected call stack.

Create a new window

Brings up a new Main View window for the selected process or thread.

Goto

Opens a dialog box that allows you to enter the name of a thread on which focus should be switched. This is useful when multiple threads, all at the same location, are collapsed into a single line. While **Change focus to this entry** always takes you to the first thread, **Goto** allows you to jump to any thread.

Add

Opens a dialog in which you can select from a list of process ids. Selecting a process id (PID) in the dialog and pressing OK will cause the process to be attached and added into the **Multiprocess View** window's list of processes.

Remove

After you select a process/thread (by highlighting it), click on **Remove** to remove it from the list of processes in the **Multiprocess View** window. A process in a `sproc` share group cannot be removed from the process group.

Ada-specific Windows

This section discusses the **Task View** and **Exception View** windows that are specific to Ada code.

Task View

Select **Views > Task View** from the Main View window menu bar to call up the **Task View** window. The **Task View** window is an Ada-specific view that provides you with task and call stack information. If you do not have Ada installed on your system, the **Task View** menu option of the **Views** menu is grayed out.

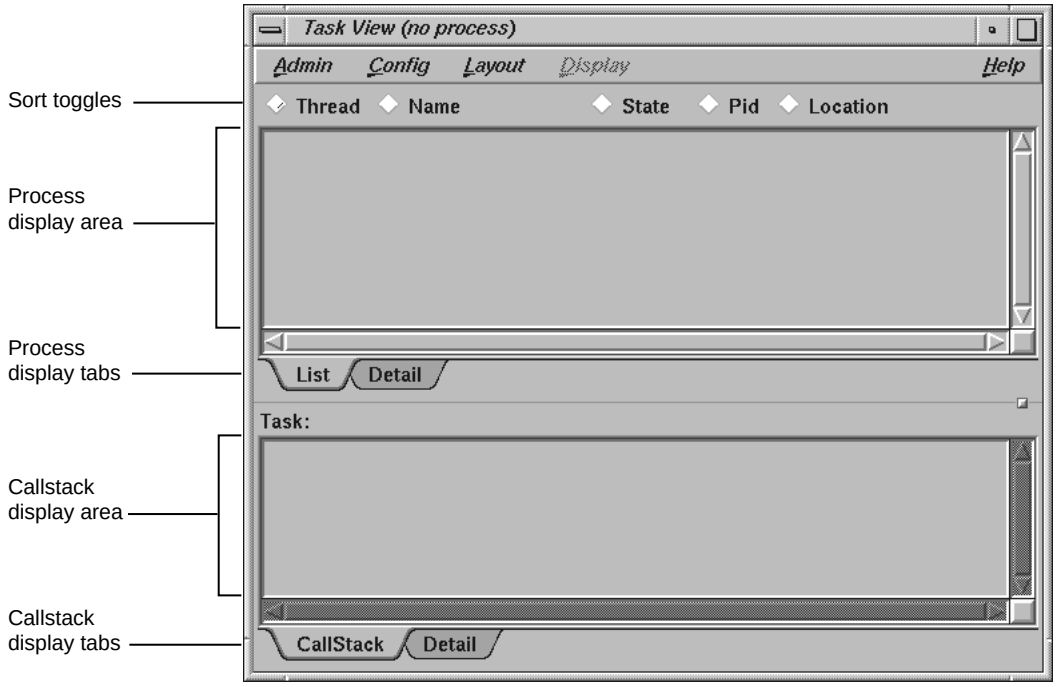

Figure A-8 Task View Window

The **Task View** menu bar contains the **Admin**, **Config**, **Layout**, **Display**, and **Help** menus. The **Help** menu is the same as that described in "Help Menu", page 196. Other menus are described in the following sections.

Admin Menu

The **Admin** menu contains the following options:

Active

This toggle activates the current window in a set of cloned windows.

Clone

Creates a clone of the current window. This action is not supported in the current release, and the option is grayed out.

Save As Text

Launches the **Save Text** dialog. This dialog allows you to save your current session as text in a file you designate.

Close

Closes the current window.

Config Menu

The **Config** menu contains the following item:

Preferences

Launches the **Task View Preference** dialog that allows you to set maximum call stack depth shown in **Task View**. Default depth is 32 frames.

Layout Menu

The **Layout** menu contains the following toggles:

Task List

Causes only the **CallStack View** to be shown.

Single Task

Causes only the **Process Display** to be shown.

Display Menu

The **Display** menu is divided into the **Task List Format** and **Callstack Format** sections. The **Task List Format** toggle buttons control which buttons appear in the toggle sort list, as well as what information is displayed in the **Process Display** area. The **Callstack Format** toggles control the amount of information to be displayed in the **Callstack Display** area of the**Task View** window.

The **Task View** Display menu contains the following toggles:

Thread/task

Displays thread/task number. This toggle is active by default.

Status

Displays process status. This toggle is active by default.

PID

Displays PID number.

Location

Displays routine name and location in the current source file.

Arg Values

Allows you to set the argument values in the **Callback Display**. This toggle is active by default.

Arg Names

Allows you to set the argument names in the **Callback Display**. This toggle is active by default.

Arg Types

Allows you to display argument types in the **Callback Display**.

PC

Allows you to set the program counter (PC) in **Task View**.

In addition to menus, **Task View** also contains the following items from which you can select to vary the display:

Sort toggles

Allows you to sort the process list by **Thread**, **Name**, **State**, **Pid**, or **Location**, depending on which of the buttons is active. Default selection is **Thread**.

Process display tabs

Allows you to view a list of tasks or details of the currently running (highlighted) task.

Callstack display tabs

Allows you to view all call stack information or call stack details of the currently selected process.

Exception View

Select **Views > Exception View** from the Main View window menu bar to display the **Exception View** window. **Exception View** is an Ada-specific view that allows you to set traps on exceptions and control exception handling. This view works only if the Ada compiler is installed. By default, this view displays only the following predefined Ada exceptions:

- Constraint errors
- Program errors
- Storage errors
- Tasking errors

In addition, a single breakpoint is set on any unhandled exception.

Figure A-9 shows a typical **Exception View** window.

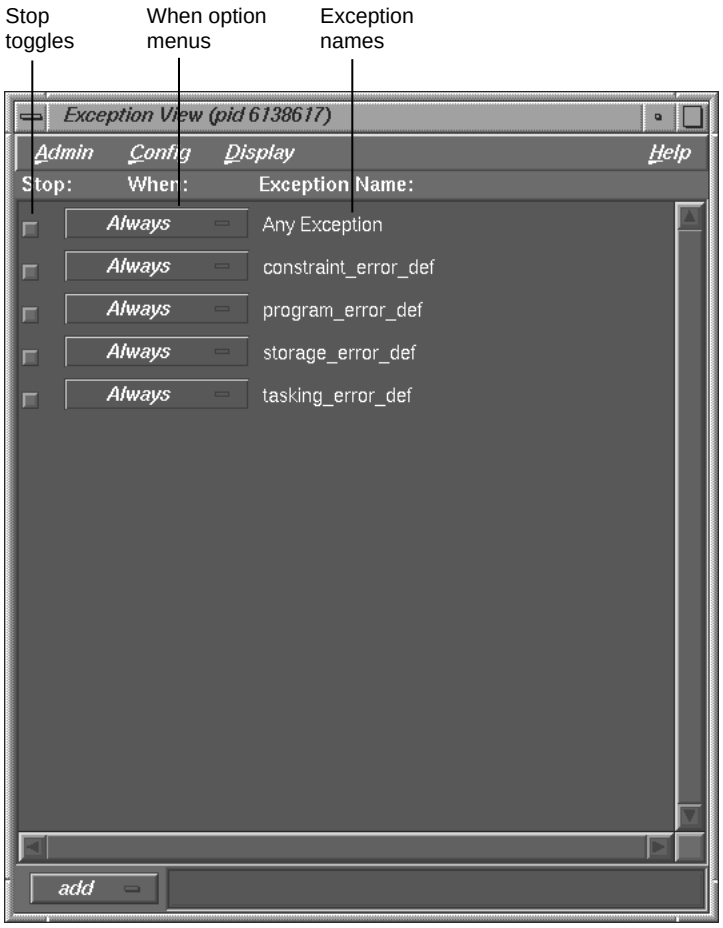

Figure A-9 Exception View

The **Admin** menu has the following options:

Active

Activates the current window in a set of cloned windows.

Clone

Creates a clone of the current window.

Save As Text

Launches the **Save Text** dialog. This dialog allows you to save your current session as text in a file you designate.

Close

Closes the current window.

The **Config** menu has the following options:

Load Exceptions

Opens the **Load User Defined Exceptions** dialog that allows you to add additional exceptions to the predefined Ada exceptions.

Save Exceptions

Opens the **Save User Defined Exceptions** dialog that allows you to save any user-defined exceptions to the predefined Ada exceptions.

The **Display** menu has the following options:

Delete All	Deletes all exception traps.
Clear All Traps	Clears all exception traps. Clearing traps is not the same as deleting traps. Clearing only temporarily affects traps while deleting removes them permanently.
Reset All Buttons	Resets all button actions.

The **Stop** boxes toggle on and off to indicate whether a trap is active.

The **When** control menus allow you to determine when an exception trap fires. The following choices are available:

Always	Stops any time the exception is raised.
WhenOthers	Stops when caught by a `when others` handler rather than an explicit handler or when unhandled.
Unhandled	Stops when the exception is unhandled.

In the un-labeled text field at the bottom right of the window you can enter a single, fully qualified Ada exception name or a single, fully qualified Ada unit name. Depending on whether the **add**, **remove**, or **find** mode is active; pressing `Enter` will cause one of the following actions to occur:

- **add** mode:
  - Single exception: Adds single exception to the exception list
  - Library unit name: Adds all exceptions found in that library unit name to the exception list
- **remove** mode:
  - Single exception: Removes single exception from the exception list
  - Library unit name: removes all exceptions found in that library unit name from the exception list
- **set** mode
- **clear** mode
- **find** mode:
  - Single exception: positions top of the exception list to single exception
  - Library unit name: positions top of the exception list to the first exception found in given library unit name

X/Motif Analyzer Windows

The X/Motif Analyzer provides specific debugging support for X/Motif applications. There are various examiners for different X/Motif objects, such as widgets and X Window System graphics context, that might be difficult or impossible to inspect using ordinary debugger functionality. See Chapter 11, "X/Motif Analyzer", page 149 for a comprehensive discussion and tutorial regarding the X/Motif Analyzer.

To access the **X/Motif Analyzer** window, pull down the **Views** menu and select **X/Motif Analyzer** (see Figure A-10, page 215).

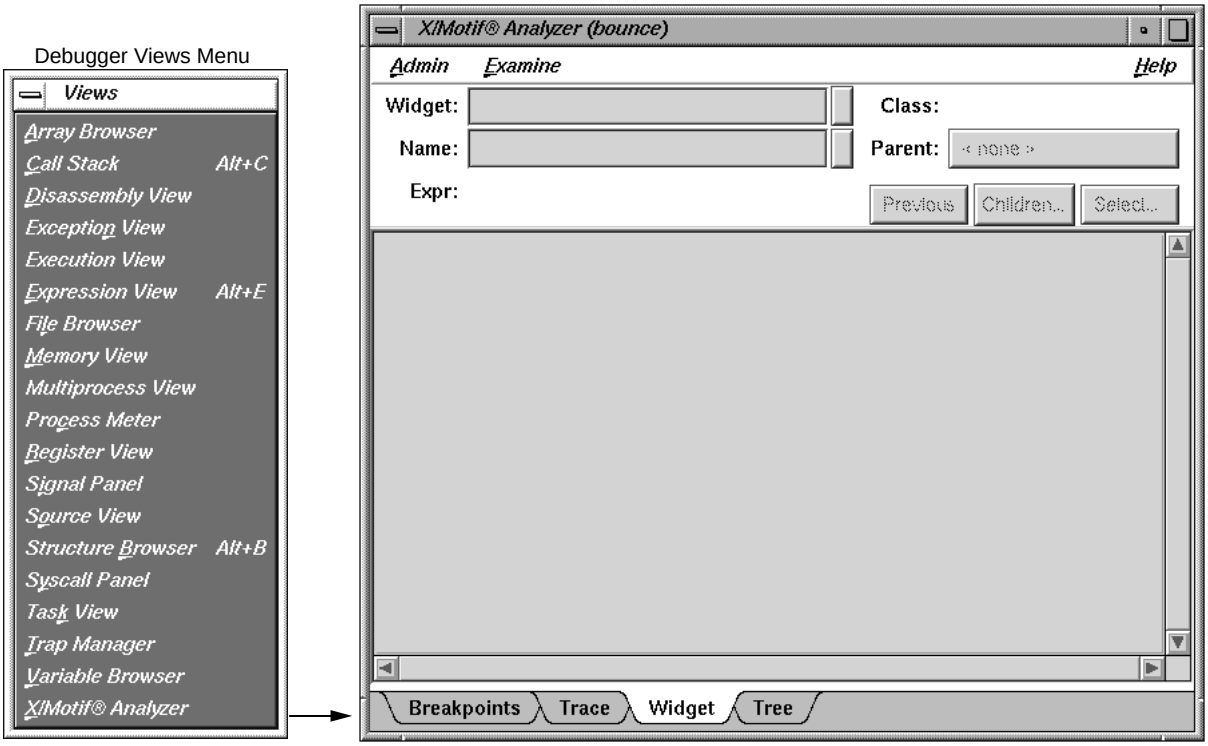

Figure A-10 Launching the **X/Motif Analyzer** Window

Global Objects

Though the X/Motif Analyzer is made up of several different examiner windows, a number of objects, such as the **Admin** menu, **Examine** menu, **Help** menu and several text bars, remain constant throughout window changes. The following examiners are available and discussed in the sections indicated:

- "Breakpoints Examiner", page 218
- "Trace Examiner", page 235
- "Widget Examiner", page 237
- "Tree Examiner", page 238

Admin Menu

The **Admin** menu offers the following menu selections:

Active

Activates the current window in a set of cloned windows. In the current release, this toggle is always active.

Clone

Creates a clone of the current window. This action is not supported in the current release and the option is grayed out.

Save As Text

Launches the **Save Text** dialog. This dialog allows you to save your current session as text in a file you designate. This selection is not available for examiners that are graphical displays, such as the Breakpoints Examiner, the Tree Examiner, and the Pixmap Examiner.

Close

Closes the current window.

Examine Menu

The **Examine** menu offers the following options:

Selection

Selects the currently highlighted object for examination. You must first highlight the name of an object before you select this option.

Widget

Uses the current selection as input to the widget examiner, then opens that examiner (see "Widget Examiner", page 237, for information).

Widget Tree

Switches the window view to the widget tree examiner (see "Tree Examiner", page 238, for information).

Widget Class

Switches the window view to the widget class examiner (see "Widget Class Examiner", page 246, for information).

Window

Switches the window view to the window examiner (see "Window Examiner", page 242, for information).

X Event

Switches the window view to the X Event examiner (see "Event Examiner", page 243, for information).

X Graphics Context

Switches the window view to the X graphics context examiner (see "Graphics Context Examiner", page 244, for information).

X Pixmap

Switches the window view to the X pixmap examiner (see "Pixmap Examiner", page 245, for information).

Examiner Tabs

In addition to access through the **Examine** menu, each examiner can be accessed through a tab at the bottom of each view (see Figure A-11).

Figure A-11 Examiner Tabs

When first launched, the X/Motif Analyzer has the following four tabs from left-to-right:

- **Breakpoints**
- **Trace**
- **Widget**
- **Tree**

As you select other examiners through the **Examine** menu, new tabs are added for the new examiners.

To delete a tab:

1. Select the tab you want to delete.
2. Right-click and select **Remove Examiner** from the pop-up menu.

   The selected tab will disappear.

   Note: You can not remove the first four tabs.

The X/Motif Analyzer will also bring up new examiner windows whenever they are needed (see Chapter 11, "X/Motif Analyzer", page 149 for more information).

Click on the collapsed tabs to the right to display them.

Return Button

Both the **Widget** and **Name** text fields have return buttons (see Figure A-12, page 220) just to the right. Clicking these buttons causes the X/Motif Analyzer to respond exactly as if you had pressed `Return` on your keyboard.Only the **Breakpoint Examiner** and **Widget Examiner** have a **Return** button.

Breakpoints Examiner

The Breakpoints examiner is not really an examiner, but a control area where you can set widget-level breakpoints. The breakpoints examiner is divided into three areas (see Figure A-12, page 220):

- The widget specification area that contains the same information as that in the Widget examiner. You can select a widget address, name, or class in this area, as well as move to the widgets parents or children, or select a widget in the application. In cases where the breakpoint type does not apply to widgets (for example, input-handler breakpoints), this area is blank.
- The parameter specification area, the contents of which vary according to the type of breakpoint you are setting. For example, for callback breakpoints, this area contains the callback name and client data; for event-handler breakpoints, it contains the event type and the client data, and so on.
- The breakpoint area, which contains the breakpoint name, a search field, and the **Add**, **Modify**, **Delete**, and **Step To** buttons. Since the **Search** text field and the four buttons appearing in the **Breakpoints** area function the same way no matter which **Breakpoint Type** is selected, descriptions for these items are included here, and will not be described in each of the remaining subsections. These are:

  Search text field

  Allows you to perform a text search through your breakpoints.

  Add button

  Allows you to add a new breakpoint.

  Modify button

  Allows you to change the selected breakpoint's settings.

  Delete button

  Deletes the selected breakpoint.

  Step To button

  Allows you to step to the next condition. **Step To** creates a temporary breakpoint, resumes the process, and waits until the process stops. This temporary breakpoint acts exactly like an ordinary breakpoint, save that the **Step To** button automatically

resumes the process and puts up a busy cursor until the condition becomes true.

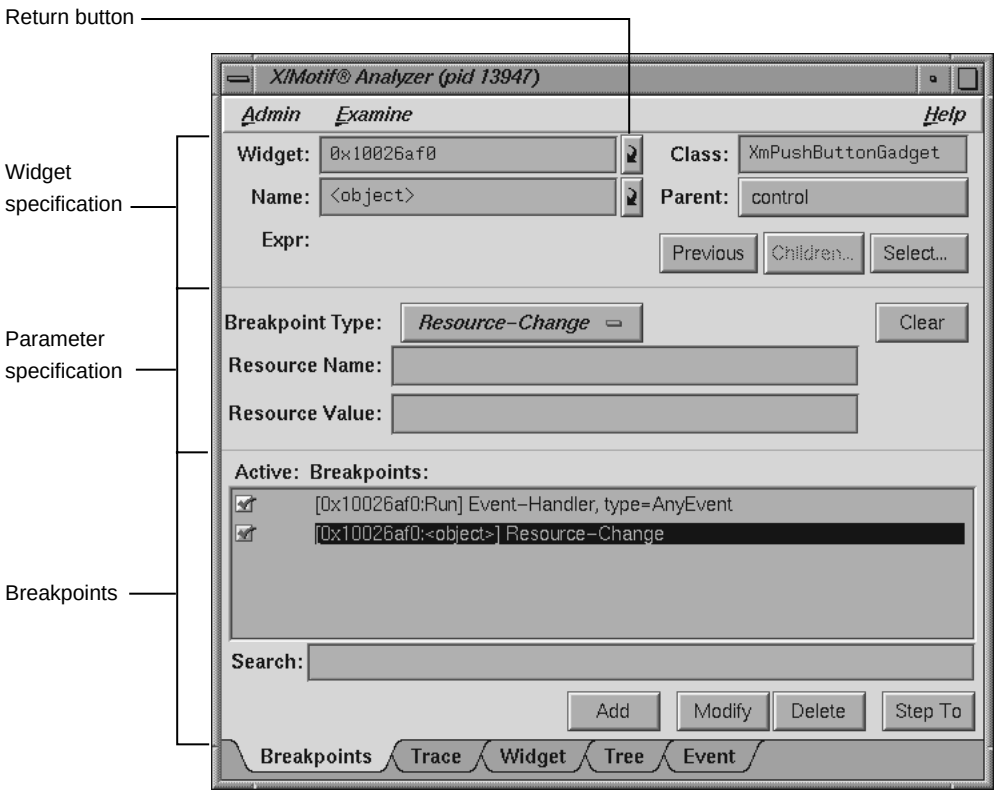

Figure A-12 Breakpoints Examiner Display in the **X/Motif Analyzer** Window

The control area has eight different breakpoint types that it can examine. These types are set through the **Breakpoint Type** options. The following **Breakpoint Type** options are available:

Callback

Widget callback installed by `XtAddCallback`. Parameters include callback name and `client_data` `XtPointer` value. See "Callback Breakpoints Examiner", page 222, for more information.

Event-Handler

Widget event handler installed by `XtAddEventHandler`. Parameters include X event type and `client_data` `XtPointer` value. See "Event-Handler Breakpoints Examiner", page 224, for more information.

Resource-Change

Resource change caused by `XtSetValues` or `XtVaSetValues`. Parameters include resource name and resource value, both strings. See "Resource-Change Breakpoints Examiner", page 226, for more information.

Timeout-Procedure

Timeout callback installed by `XtAppAddTimeOut`. Parameters include `client_data` `XtPointer` value. See "Timeout-Procedure Breakpoints Examiner", page 227, for more information.

Input-Handler

Input callback installed by `XtAppAddInput`. Parameters include `client_data` `XtPointer` value. See "Input-Handler Breakpoints Examiner", page 228, for more information.

State-Change

Various widget state changes (for example, managed or realized). Parameters include widget state. See "State-Change Breakpoints Examiner", page 230, for more information.

X-Event

X event received by target application. Parameters include X event type. See "X-Event Breakpoints Examiner", page 232, for more information.

X-Request

X request received by target application. Parameters include X request type. See "X-Request Breakpoints Examiner", page 233, for more information.

Callback Breakpoints Examiner

When the **Callback** option of the **Breakpoint Type** option button in the **Breakpoints Examiner** is selected, the Callback Breakpoints Examiner is displayed.

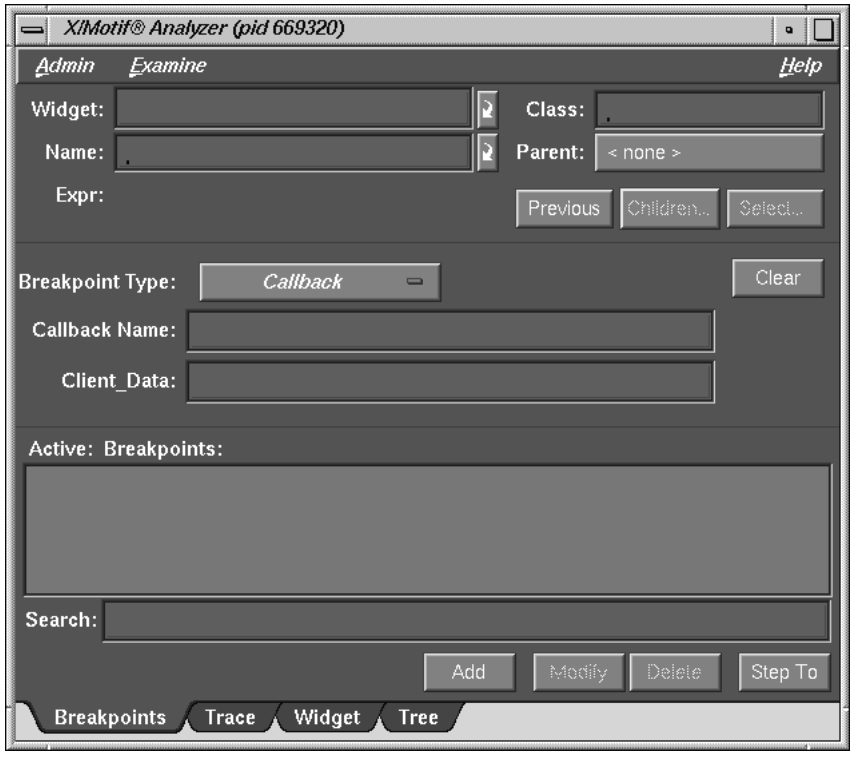

Figure A-13 Callback Breakpoints Examiner

The Callback Breakpoints examiner contains the following items:

Widget text field

Allows you to choose a widget to examine by entering the widget address.

Name text field

Allows you to choose a widget to examine by entering the widget name.

Class text field

Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.

Parent text field

Allows you to move to the parent of the currently selected widget.

Previous button

Moves you to the previously selected widget.

Children button

Shows you the widget's children (it is grayed out if the selected widget cannot have children).

Select button

Allows you to select the widget in the target process.

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **Callback** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Callback Name text field

Allows you to set the **Name** of the callback for the breakpoint.

Client_Data text field

Allows you to pass and get back pointer values for **Client_Data**.

Event-Handler Breakpoints Examiner

When the **Event-Handler** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-14, page 224.

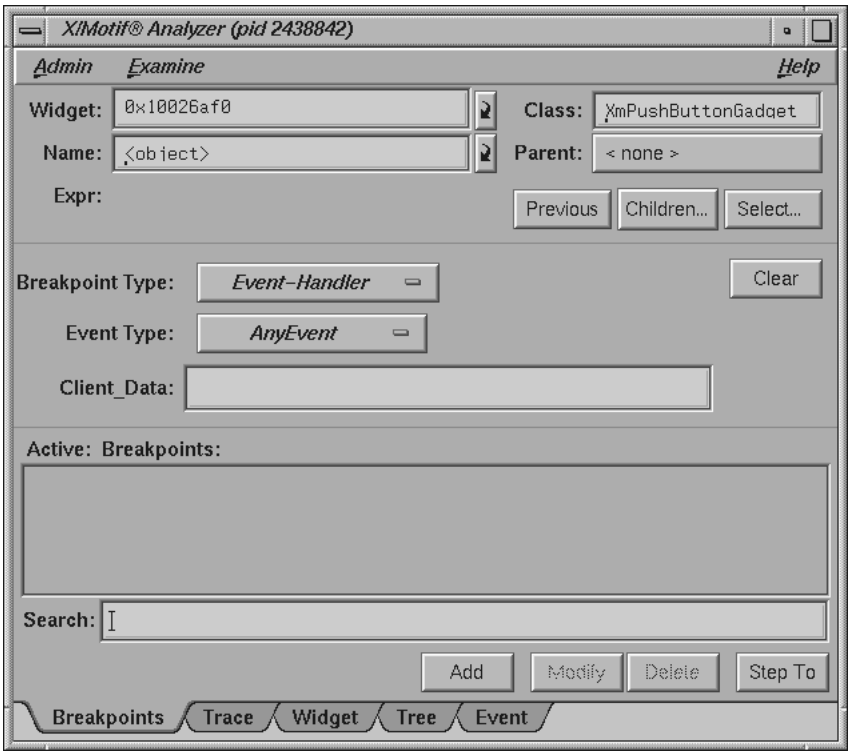

Figure A-14 Event-Handler Breakpoints Examiner

The Event-Handler Breakpoints examiner contains the following items:

Widget text field

Allows you to choose a widget to examine by entering the widget address.

Name text field

Allows you to choose a widget to examine by entering the widget name.

Class text field

Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.

Parent text field

Allows you to move the parent of the currently selected widget.

Previous button

Moves you to the previously selected widget.

Children button

Shows you the widget's children (it is grayed out if the selected widget cannot have children).

Select button

Allows you to select the widget in the target process.

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **Event-Handler** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Event Type option button

Allows you to set the event type for a given breakpoint.

Client_Data text field

Allows you to pass and get back pointer values for the **Client_Data**.

Resource-Change Breakpoints Examiner

When the **Resource-Change** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-12, page 220.

The Resource-Change Breakpoints examiner contains the following items:

Widget text field

Allows you to choose a widget to examine by entering the widget address.

Name text field

Allows you to choose a widget to examine by entering the widget name.

Class text field

Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.

Parent text field

Allows you to move the parent of the currently selected widget.

Previous button

Moves you to the previously selected widget.

Children button

Shows you the widget's children (it is grayed out if the selected widget cannot have children).

Select button

Allows you to select the widget in the target process.

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **Resource-Change** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Resource Name text field

Allows you to set the resource name for the breakpoint.

Resource Value text field

Allows you to set the resource value for the breakpoint.

Timeout-Procedure Breakpoints Examiner

When the **Timeout Procedure** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-15.

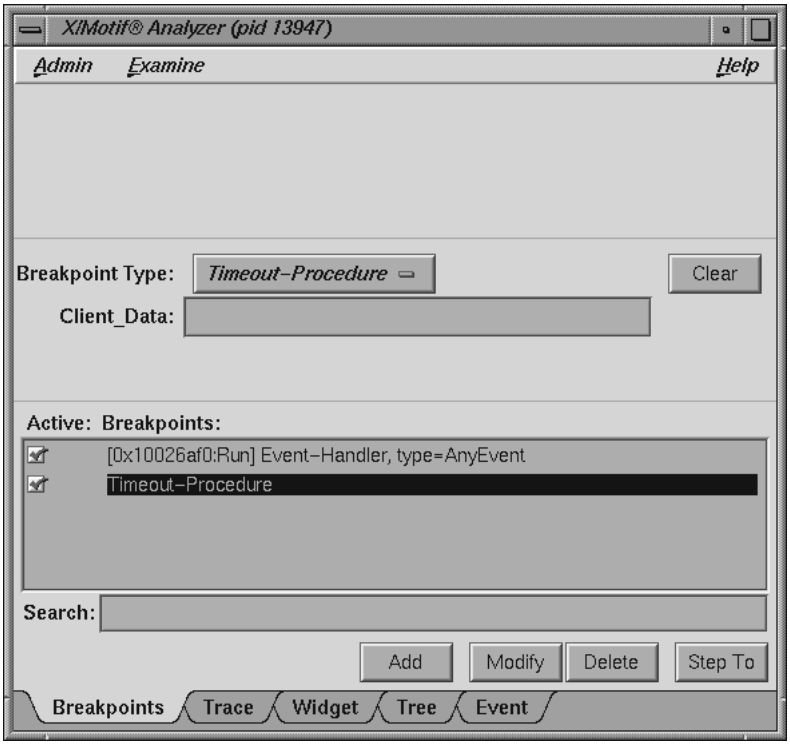

Figure A-15 Timeout-Procedure Breakpoints Examiner

The Timeout-Procedure Breakpoints examiner contains the following items:

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **Timeout-Procedure** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Client_Data text field

Allows you to pass in and get back pointer values for the **Client_Data**.

Input-Handler Breakpoints Examiner

When the **Input-Handler** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-16.

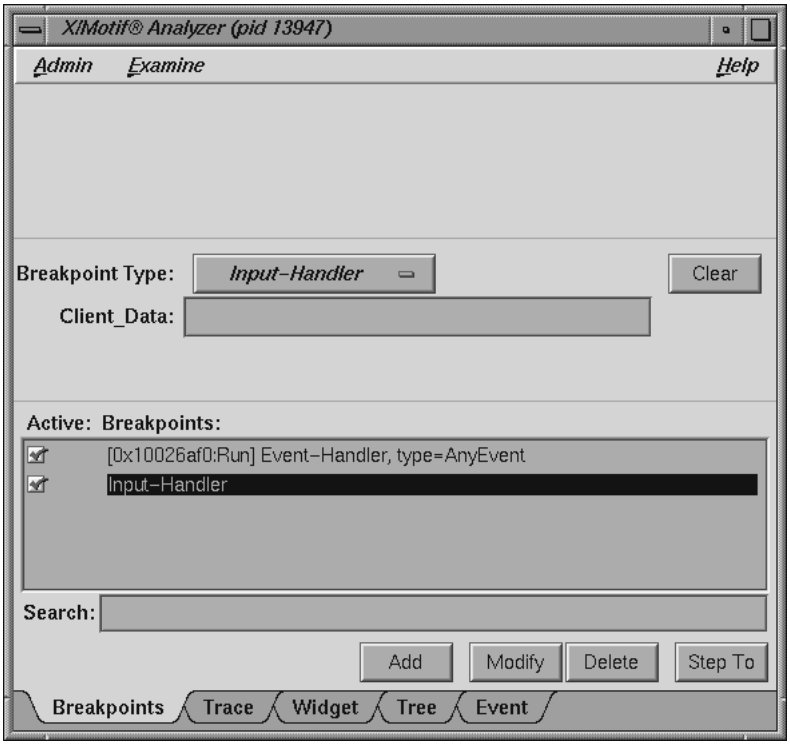

Figure A-16 Input-Handler Breakpoints Examiner

The Input-Handler Breakpoints examiner contains the following items:

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **Input-Handler** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Client_Data text field

Allows you to pass in and get back pointer values for the **Client_Data**.

State-Change Breakpoints Examiner

When the **State-Change** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-17.

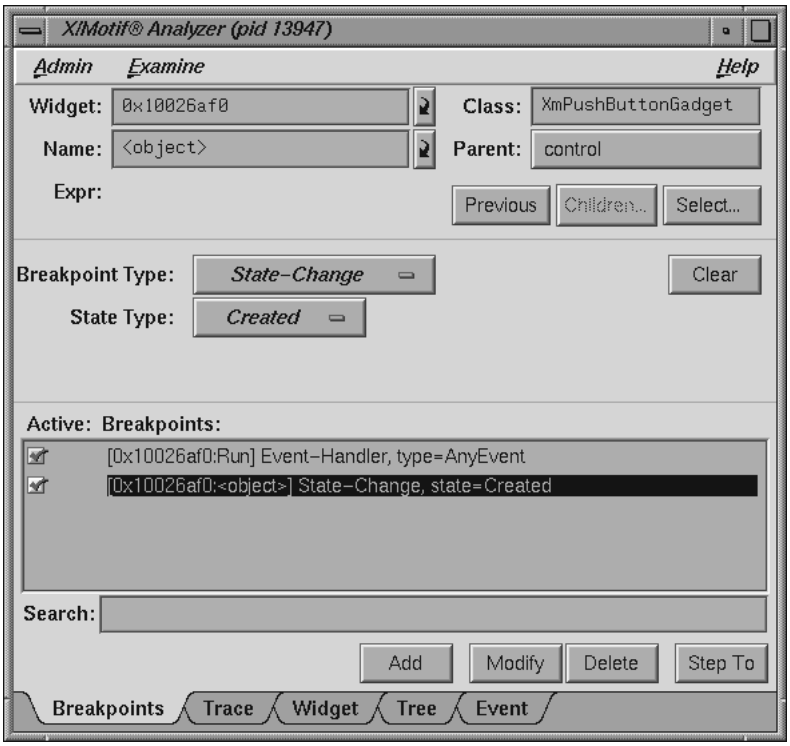

Figure A-17 State-Change Breakpoints Examiner

The State-Change Breakpoints examiner contains the following items:

Widget text field

Allows you to choose a widget to examine by entering the widget address.

Name text field

Allows you to choose a widget to examine by entering the widget name.

Class text field

Allows you to choose a widget to examine by entering the widget's class. Leave the field blank or enter **Any** to select all widgets.

Parent button

Allows you to move the parent of the currently selected widget.

Previous button

Moves you to the previously selected widget.

Children button

Shows you the widget's children (it is grayed out if the selected widget cannot have children).

Select button

Allows you to select the widget in the target process.

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **State-Change** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

State Type option button

Allows you to set the state change type for a given breakpoint.

X-Event Breakpoints Examiner

When you select the **X-Event** option of the **Breakpoint Type** option button in the Breakpoint Examiner, the examiner appears as shown in Figure A-18, page 232.

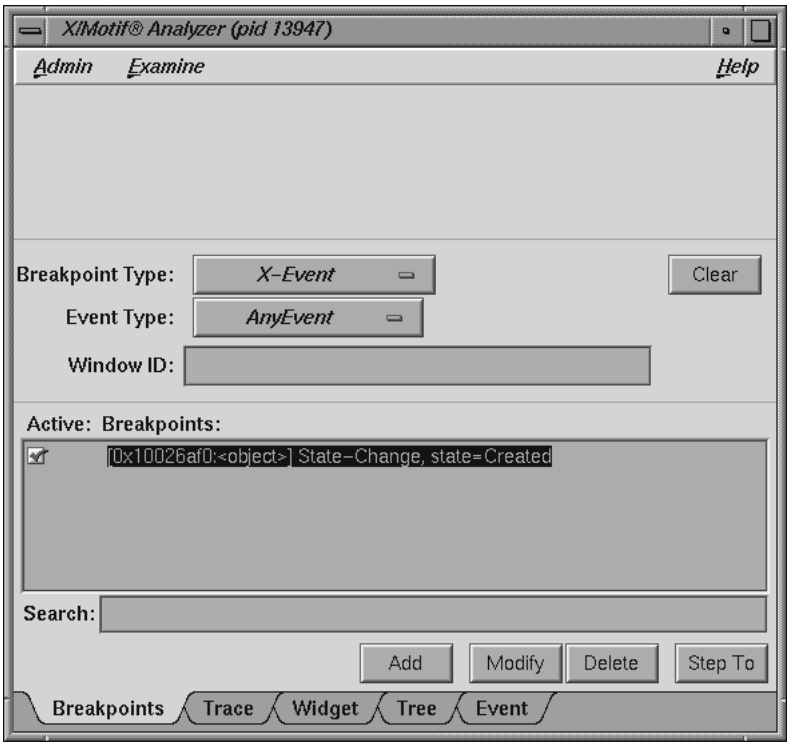

Figure A-18 X-Event Breakpoints Examiner

The X-Event Breakpoints examiner contains the following items:

Breakpoint Type button

Allows you to select the type of breakpoint you wish to set. In this section, **X-Event** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Event Type button

Allows you to set the event type for a given breakpoint.

Window ID text field

Allows you to set the Window ID value for the breakpoint.

X-Request Breakpoints Examiner

When the **X-Request** option of the **Breakpoint Type** option button in the Breakpoint Examiner is selected, the examiner appears as shown in Figure A-19, page 234.

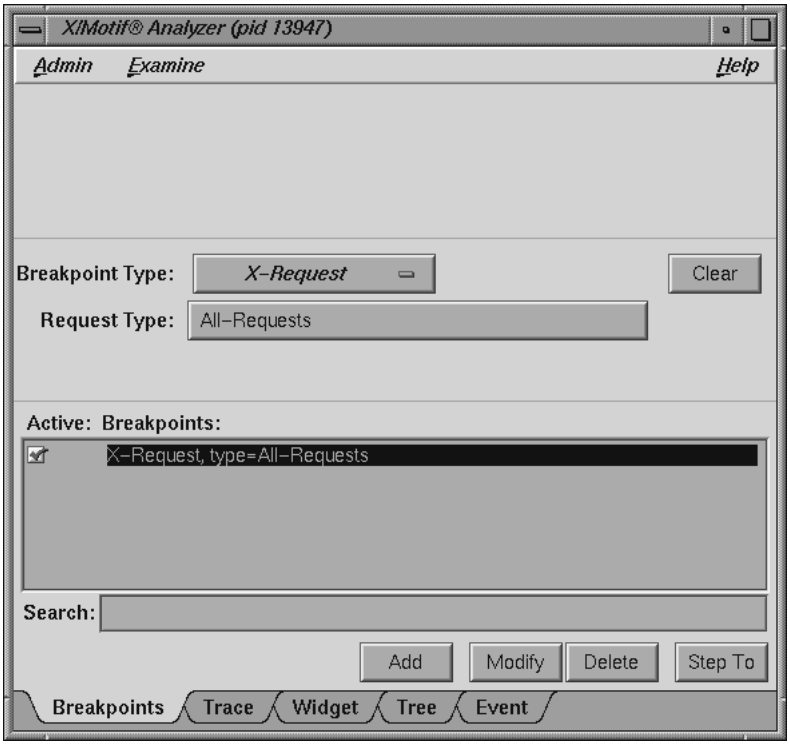

Figure A-19 X-Request Breakpoints Examiner

The X-Request Breakpoints examiner contains the following items:

Breakpoint Type option button

Allows you to select the type of breakpoint you wish to set. In this section, **X-Request** is selected.

Clear button

Clears all the current breakpoint selections and text fields.

Request Type button

Launches the **Request Type Selection** dialog (see Figure A-20). This dialog allows you to select the type of X-Request used for your breakpoint. The information displayed is in outline form; selecting a

given item selects all its subitems. For example, if you select `Window-Category`, `CreateWindow`, `ChangeWindowAttributes`, `GetWindowAttributes`, and so on are also selected.

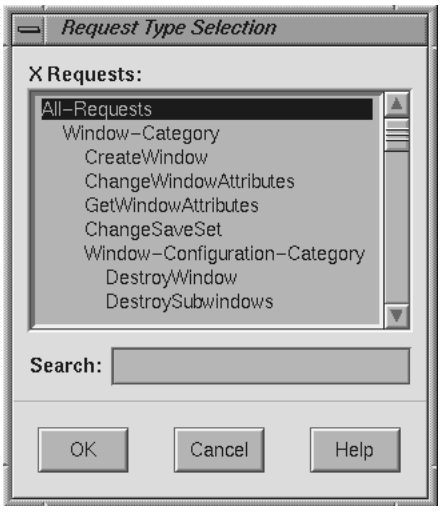

Figure A-20 Request Type Selection Dialog

Trace Examiner

The Trace examiner (see Figure A-21, page 236) is a control area where you can trace the execution of your application and collect the following date:

- X Server Events
- X Server Requests
- Widget Event Dispatch Information
- Widget Resource Changes (through `XtSetValues`)
- Widget State Changes (`create`, `destroy`, `manage`, `realize`, and `unmanage`)
- **Xt** Callbacks (`widget`, `event handler`, `work proc`, `timeout`, `input`, and `signal`)

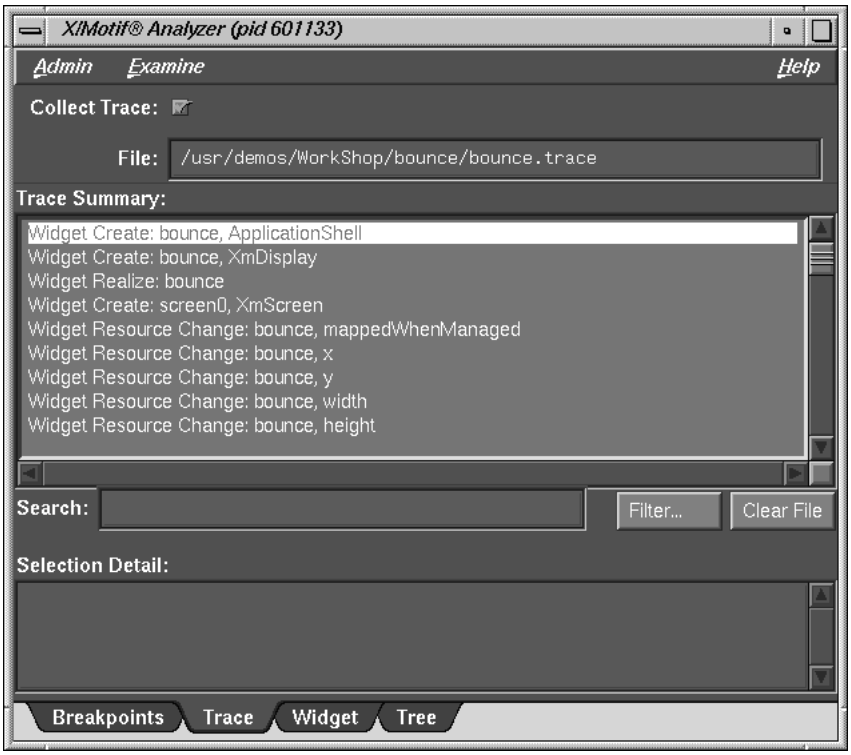

Figure A-21 Trace Examiner

The Trace examiner contains the following items:

Collect Trace toggle

Allows you to turn the tracing on and off.

File text field

Allows you to select the file name for the trace. If no file is selected, a default *filename* for the trace is chosen.

Search text field

Allows you to perform an incremental, textural search for the trace list.

Filter button

Launches a dialog that allows you to select the trace entry types you want displayed in the list.

Clear File button

Erases the trace file. Any subsequent trace information goes to the beginning of the file.

Widget Examiner

The Widget examiner (see Figure A-22) displays the internal `Xt` widget structure, as well as the `Xt` inheritance implementation using nested C constructs.

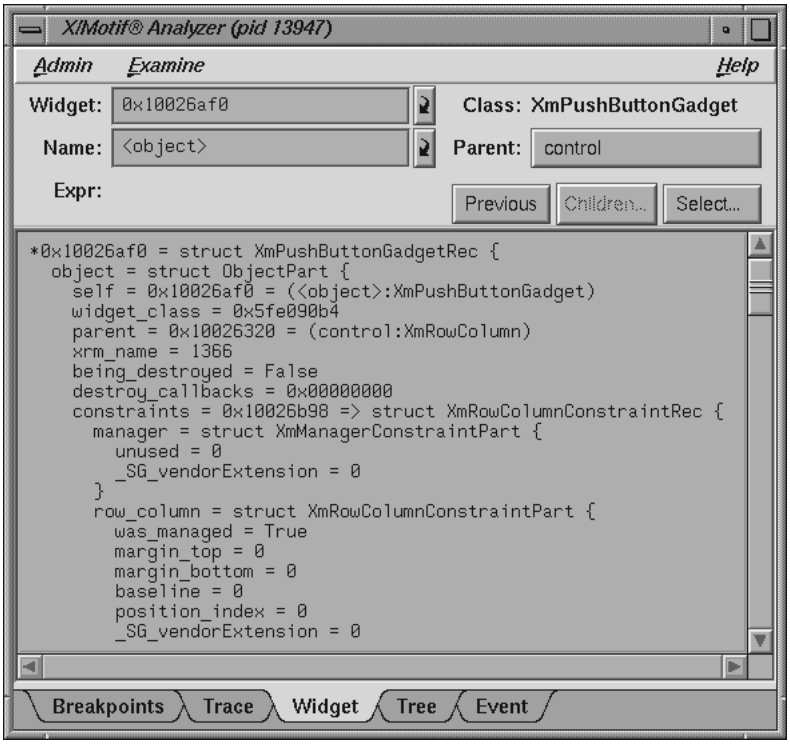

Figure A-22 Widget Examiner

The Widget examiner contains the following items:

Widget text field

Allows you to choose a widget to examine by entering the widget address.

Name text field

Allows you to choose a widget to examine by entering the widget name.

Parent button

Allows you to move the parent of the currently selected widget.

Previous button

Moves you to the previously selected widget.

Children button

Shows you the widget's children. (It is grayed out if the selected widget does not have children.)

Select button

Allows you to select the widget in the target process.

Tree Examiner

The Tree examiner (see Figure A-23, page 239) displays the widget hierarchy.

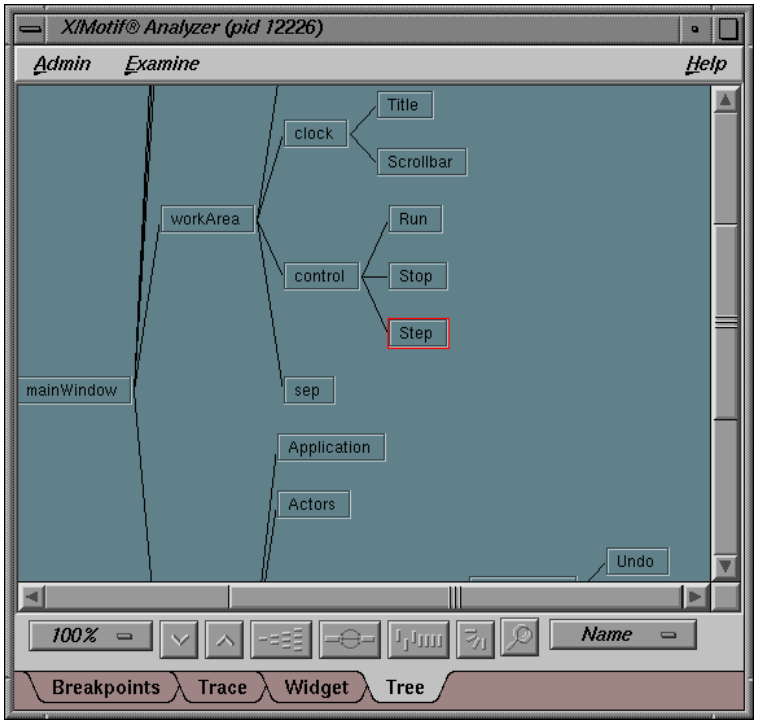

Figure A-23 Tree Examiner

You may double-click a node to view that widget in the Widget examiner.

If the Tree examiner is currently selected, it will not automatically fetch the current widget tree each time the process stops. To force retrieval of the widget tree, select another examiner and then go back to the Tree examiner. Or, click on the **Tree** tab.

The graphical buttons across the bottom of the Tree Examiner window from left-to-right have the following functions.

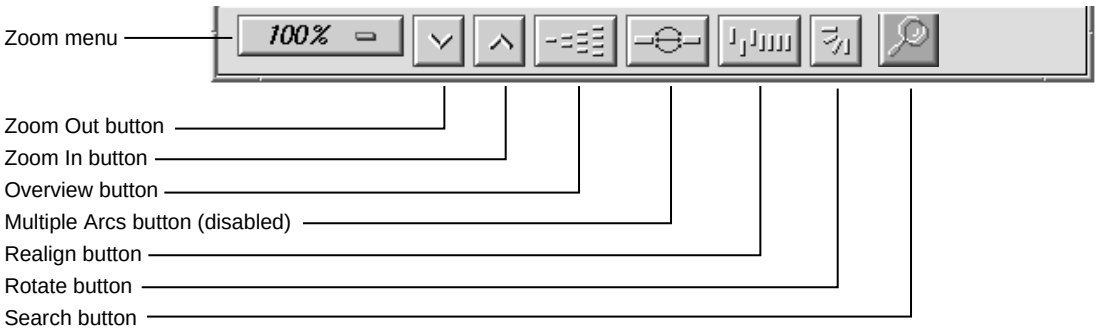

Figure A-24 Tree Examiner Window Graphical Buttons

Zoom percentage indicator

Click on this indicator to bring up a submenu from which you can select a different percentage.

Zoom Out button

Decreases the zoom percentage.

Zoom In button

Increases the zoom percentage.

Overview button

Shows the entire tree structure in another window along with an indication of which portion of the tree is currently being displayed in the full-sized Tree Examiner window.

Multiple Arcs button

This button shows/hides multiple connections between nodes on the graph. For example, if `main` calls `foo` several times, you will see a line (arc) for all calls made.

Realign button

This button resets the graph back to its original configuration (before you began working with it). Also, hidden nodes will reappear.

Rotate button

Change the format of the window from up-down to left-right, or vice-versa.

Search button

Search for text in the tree using a Widget Hierarchy Search Text window.

Widget View Type Option button

Click on this indicator to change the kind of information being displayed in the nodes of the tree. This will bring up a submenu which allows you to choose Name, Class, ID (widget address) or Window (indicate <gadget> or address).

Callback Examiner

The Callback examiner (see Figure A-25) automatically appears when the process is stopped somewhere in a callback. It first displays the callstack frame. Then it displays information about the widget in the callback. Finally, it displays the proper callback structure contained in the **call_data** argument to the callback procedure, based on the widget type and the callback name.

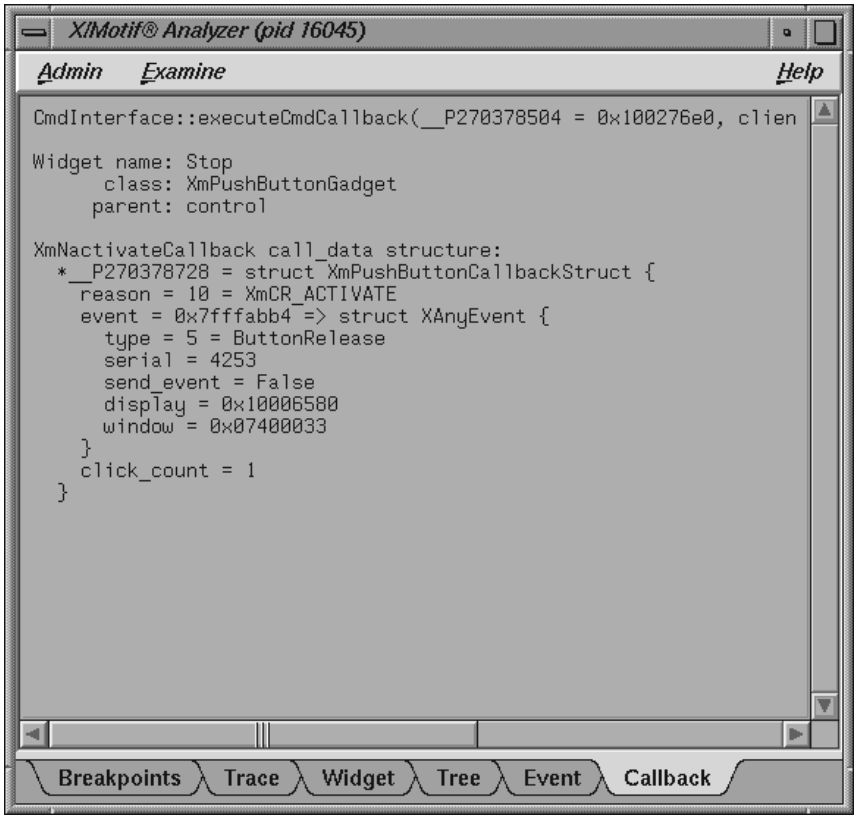

Figure A-25 Callback Examiner

Window Examiner

The Window examiner (see Figure A-26) displays window attributes for an X window and the parent and children window IDs. These attributes are returned by `XGetWindowAttributes`, with decoding of the visual structure, enums, and masks.

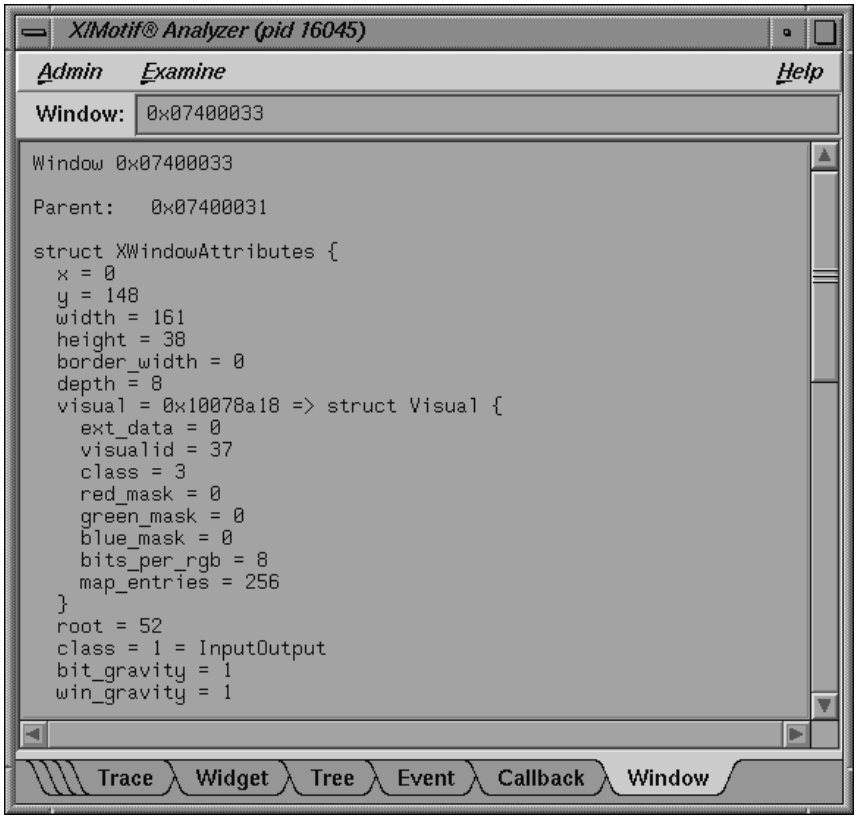

Figure A-26 Window Examiner

The Window examiner contains the **Window** text field that displays the address of the window that is being examined. You may change to a different window by entering a new address and pressing `Enter`.

Event Examiner

The Event examiner (see Figure A-27) displays the event structure for an **XEvent** pointer. The proper **XEvent** union member is used, and enums and masks are decoded.

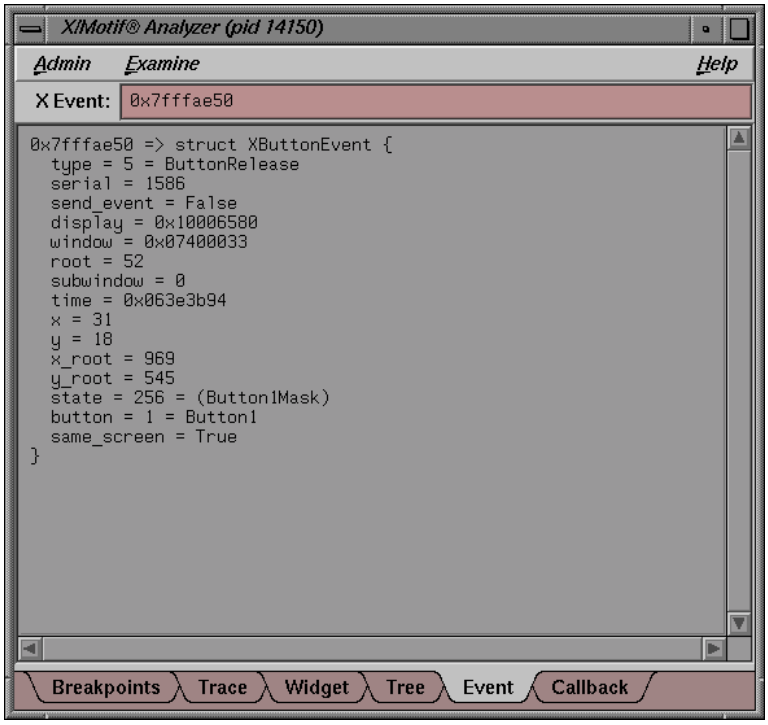

Figure A-27 Event Examiner

The Event examiner contains the **X Event** text field, which displays the address of the X event that is being examined. You may change to a different X event by entering a new address and pressing `Enter`.

Graphics Context Examiner

The Graphics Context examiner (see Figure A-28, page 245) displays the X graphics context attributes that are cached by `Xlib` in the form of an `XGCValues` structure. Enums and masks are decoded.

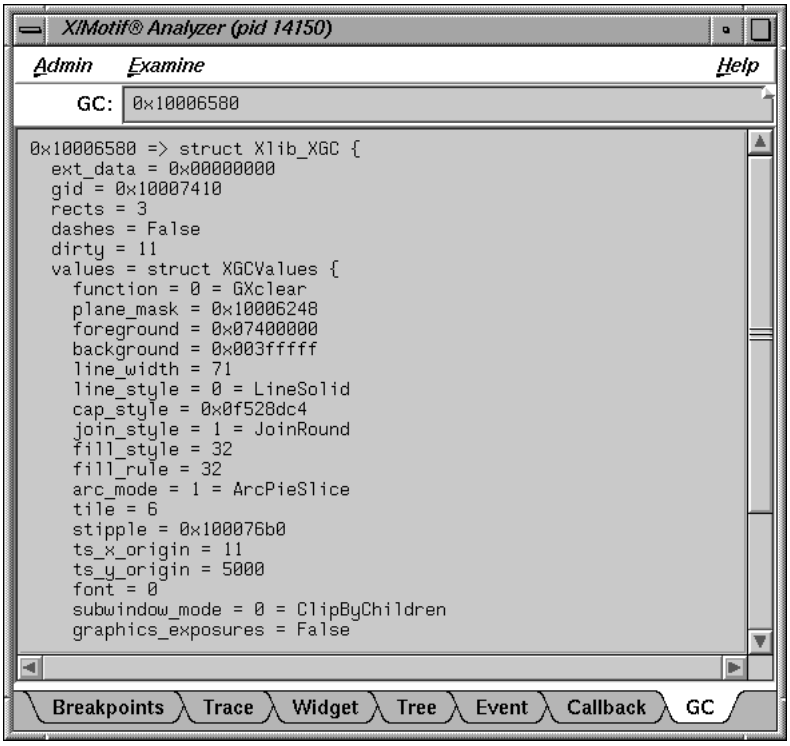

Figure A-28 Graphics Context Examiner

The Graphics Context examiner contains the **GC** text field that displays the address of the graphics context that is being examined. You may change to a different context by entering a new address and pressing `Enter`.

Pixmap Examiner

The Pixmap examiner (see Figure A-29, page 246) displays basic attributes of an X pixmap, like size and depth. It also attempts to provide an ASCII display of small pixmaps, using the units digit of the pixel values.

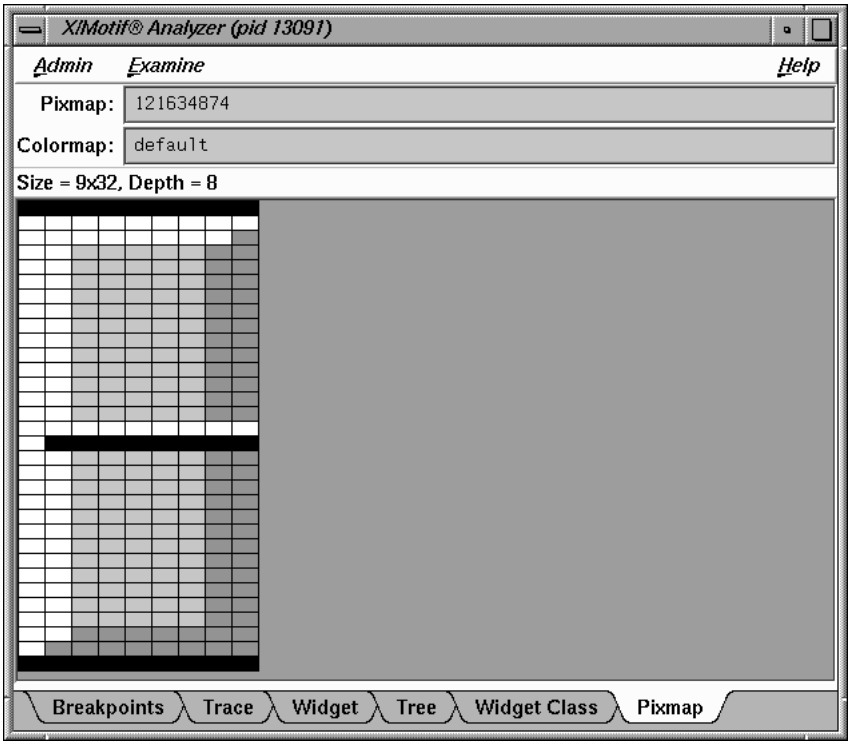

Figure A-29 Pixmap Examiner

The Pixmap examiner displays the contents of an X pixmap. To specify an X pixmap identifier, enter a numeric expression in the top text field of the window. Then, use `default` as the colormap identifier to specify the default X colormap for your screen. In the pixmap display, left-click on a pixel to see the pixel value, position, and red-green-blue intensities.

Widget Class Examiner

The Widget Class examiner (see Figure A-30, page 247) displays the `Xt` widget class structure, as well as the `Xt` inheritance implementation using nested C constructs.

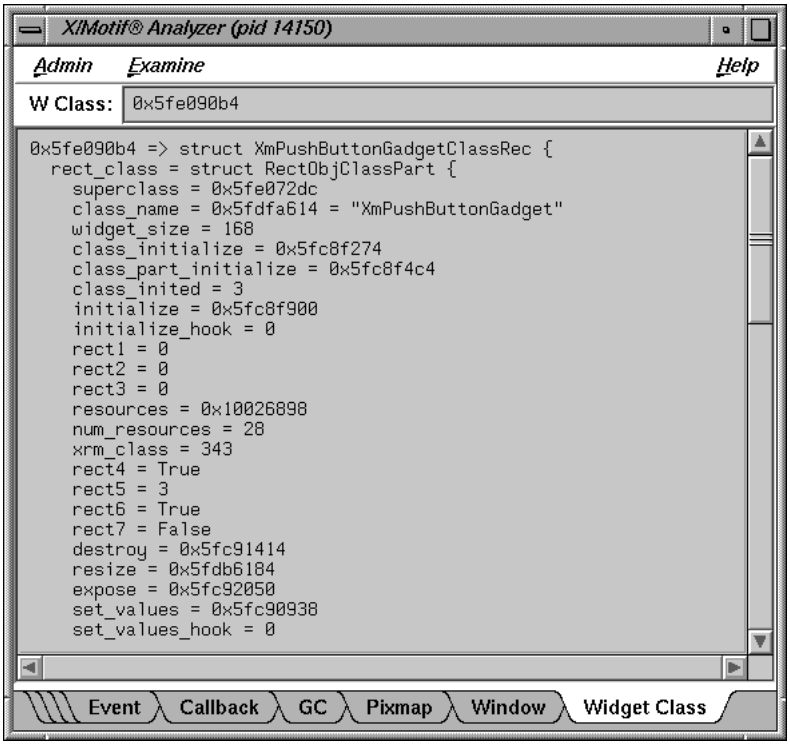

Figure A-30 Widget Class Examiner

The Widget Class examiner contains the **W Class** text field, which displays the address of the widget class that is being examined. You may change to a different widget class by entering a new address and pressing `Enter`.

Trap Manager Windows

In addition to setting traps by using the command line, the **Views** menu of the Main View window provides you with three views specific to trap management:

- **Trap Manager**
- **Signal Panel**
- **Syscall Panel**

Call up the **Trap Manager** window from the Main View window menu bar as follows:

Views
Trap Manager

Trap Manager

The **Trap Manager** allows you to set, edit, and manage traps (used in both the Debugger and Performance Analyzer). The X window is shown in Figure A-31.

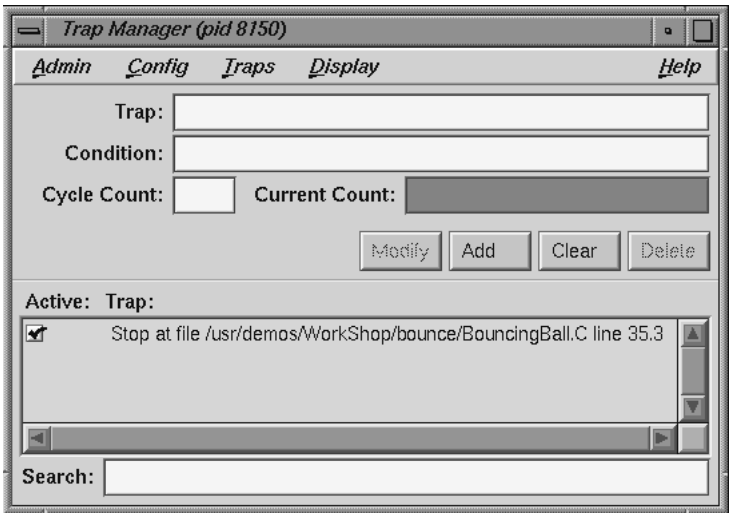

Figure A-31 Trap Manager Window

The **Trap Manager** window contains the following items (besides the menu bar, which is discussed below):

Trap text field

Contains a description of the trap.

Condition text field

Contains the condition of the trap.

Cycle Count text field

Displays the current cycle count.

Current Count text field

Displays the current trap count.

Full button

Allows you to toggle between display of full and partial path names.

Modify button

Allows you to change the selected breakpoint's settings.

Add button

Allows you to add a new breakpoint.

Clear button

Clears all the current breakpoint selections and text fields.

Delete button

Deletes the selected breakpoint.

Active label

If selected with a check mark, the trap is enabled.

Trap display area

Contains a description of each trap, and a toggle to indicate whether or not the trap is active.

Search text field

Allows you to perform an incremental textual search for the trap list.

The **Trap Manager** window has a menu bar which contains the **Admin**, **Config**, **Traps**, **Display**, and **Help** menus. The **Admin** menu is the same as that described in "Admin Menu", page 208. The **Help** menu is the same as that described in "Help Menu", page 196. The other menus are described in the following sections.

Config Menu

The **Config** menu contains the following items:

Load Traps

Brings up the **File** dialog allowing you to load the traps from a file.

Save Traps

Brings up the **File** dialog allowing you to save the current traps to a file.

Traps Menu

The **Traps** menu has options that allow you to set traps under a number of conditions. The following conditions are available:

At Source Line

Highlight a line in the Main View window's source pane before selecting this option to set a breakpoint at the selected line.

Entry Function

Highlight a function name in the Main View window's source pane before selecting this option to set a breakpoint at the entry to the function.

Exit Function

Highlight a function name in the Main View window's source pane before selection this option to set a breakpoint at the exit from the function.

Stop Trap Default

Causes a trap created to be a "stop" trap. Toggles with the **Sample Trap Default**.

Sample Trap Default

Causes a trap created to be a "sample" trap. Toggles with the **Stop Trap Default**.

Group Trap Default

ON/OFF toggle to cause a trap created to have the "pgrp" option.

Stop All Default

ON/OFF toggle to cause a trap created to have the **All** option.

Display Menu

The **Display** menu contains the following item:

Delete All

Deletes all traps from the trap list.

Signal Panel

The **Signal Panel** displays the signals that can occur. You can specify which signals trigger traps and which are to be ignored. The **Signal Panel** is shown in Figure A-32, page 252.

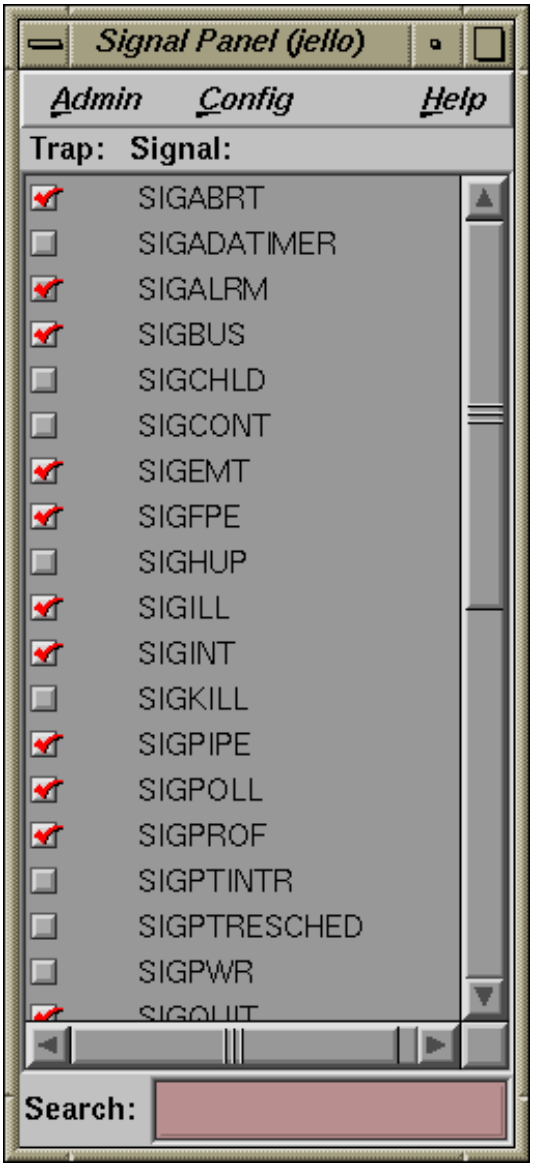

Figure A-32 Signal Panel

The **Signal Panel** contains an **Admin** menu (described in "Admin Menu", page 208) and a **Help** menu (described in "Help Menu", page 196). Each signal trigger trap in the display has a toggle associated with it. In addition, the panel has a **Search** text field.

Note: When debugging IRIX 6.5 pthreads, the **Signal Panel** is inaccessible if more than one thread is active.

Syscall Panel

The **Syscall Panel** allows you to set traps at the entry to or exit from system calls. The **Syscall Panel** is shown in Figure A-33, page 254.

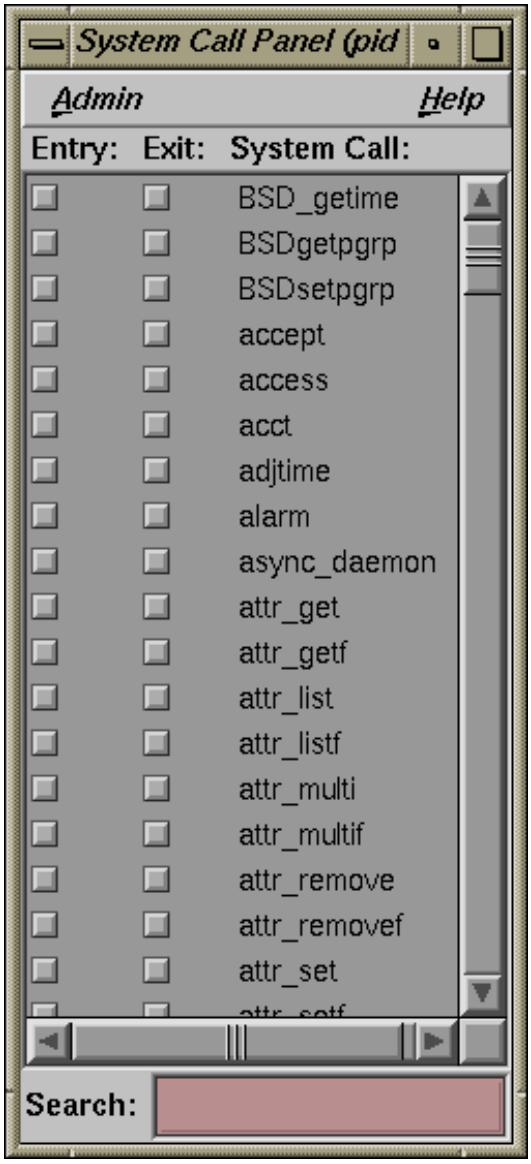

Figure A-33 Syscall Panel

The **Syscall Panel** contains an **Admin** menu (described in "Admin Menu", page 208) and a **Help** menu (described in "Help Menu", page 196). Each system call in the display has two toggles associated with it: one to set a trap on entry, one to set a trap on exit. In addition, the panel has a **Search** text field.

Data Examination Windows

There are several windows that are used primarily to examine your program's data:

- "Array Browser Window", page 255
- "Call Stack Window", page 271
- "Expression View Window", page 273
- "File Browser Window", page 276
- "Structure Browser Window", page 277
- "Variable Browser Window", page 287

Array Browser Window

To examine numeric, pointer, or character string data in an array variable, select **Array Browser** from the **Views** menu at a point in the process where the variable is present. The **Array Browser** allows you to view elements in a multi-dimensional array (up to 100 x 100 elements), presented in a spreadsheet and graphically, if desired.

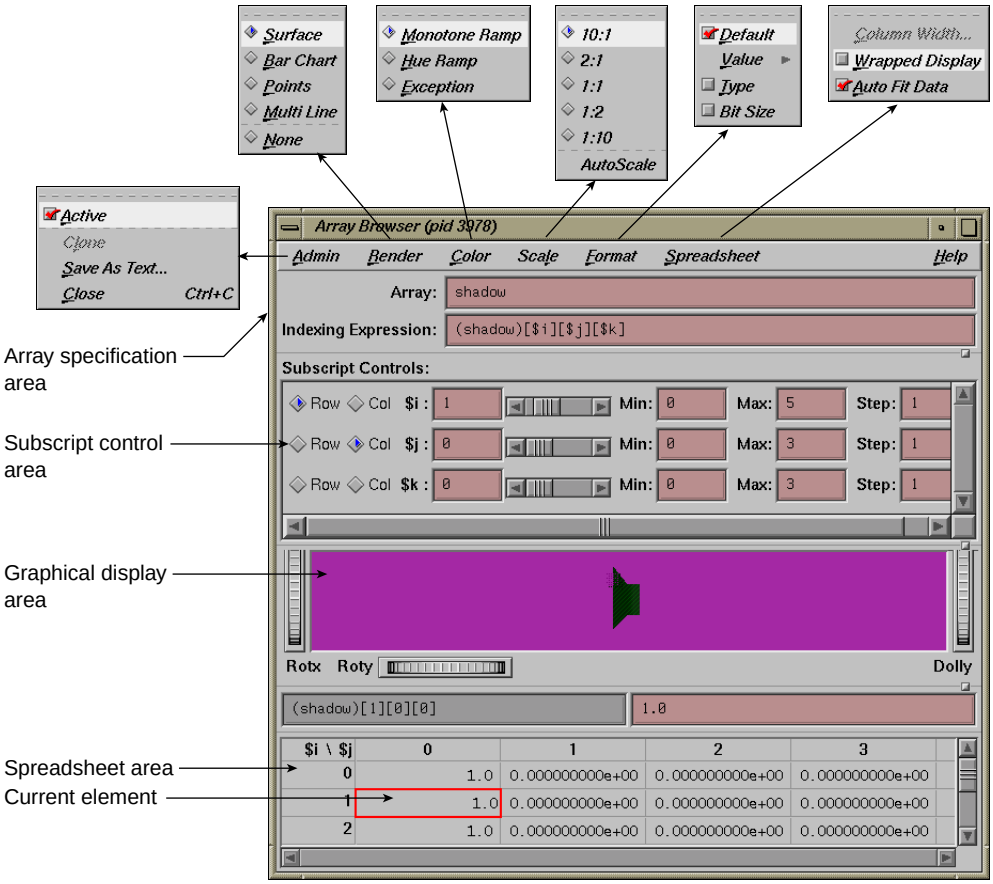

Figure A-34 Array Browser with **Display** Menu Options

Note: The **Render**, **Color**, and **Scale** tear-off menus are available only if you are have an SGI workstation with Open Inventor installed on it.

The *array specification area* allows you to specify the variable and its dimensions. It consists of the following fields:

Array

Allows you to enter the name of the array variable. This entry is language-dependent.

For Fortran, the expression may be an array or a dummy array variable name. If the last dimension of the array is unspecified (*), a subscript value of 1 is assumed initially.

For C and C++, the entry may be an array or a pointer. If pointers are used, the expression is treated as though it were a single element, in which case you need to use the subscript controls to see more than the first element.

Indexing Expression

The expression used to view an element in the array. It is filled in automatically when you specify an array to view.

The expression supplied is language-specific. It represents the indexing expression used in the language to access a particular element. The subscripts are specified by special indexing variables (`$i`, `$j`, `$k`, and so forth) that can be manipulated in the subscript controls area.

The **Subscript Controls** area allows you to control which elements in the variable are displayed and allows you to shift the current element. The number of dimensions in the array governs the number of controls that are displayed. A close-up view of the subscript controls area appears in Figure A-35.

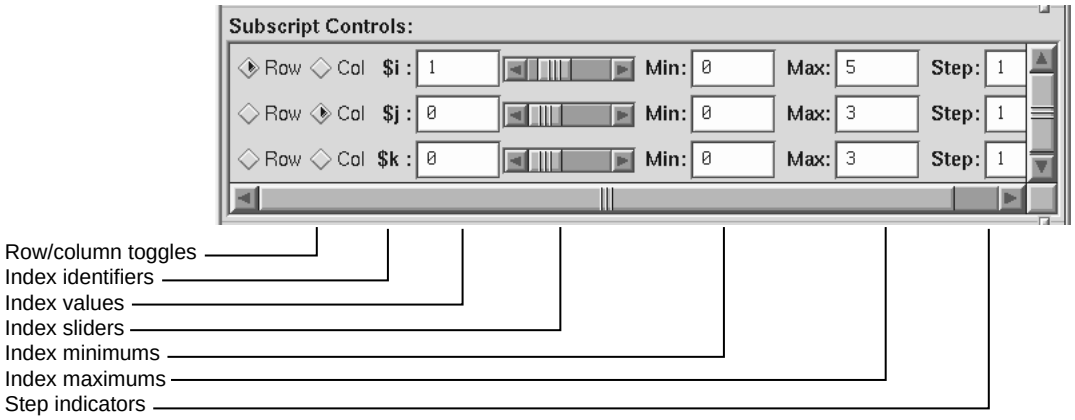

Figure A-35 Subscript Controls Area in the **Array Browser**

The **Subscript Controls** area provides the following features:

Row/column toggles

Controls whether an index variable represents rows or columns (or neither) in the spreadsheet area. You are not limited by the number of dimensions of an array, but you can only view a two-dimensional orthogonal slice of the array at a time.

Index identifiers

Indicates to which subscript the controls in the row refer.

Index values

Shows the value of the subscript for the element currently in the focus cell. You can enter a different value if you wish.

Index sliders

Lets you move the focus cell along the particular dimension.

Index minimums

Identifies the beginning visible element in that particular dimension.

Index maximums

Identifies the last visible element in that particular dimension. If you have an unspecified array, you can use this field to specify the last element in the vector to be displayed in the spreadsheet.

Step indicators

Specifies the increment between adjacent elements in the dimension to be displayed. A value of 1 displays consecutive data. Specifying some *n* greater than 1 allows you to display every *n* element in a vector.

Control area scroll bars

Allows you to expose hidden portions of the subscript control area if your window is not large enough for viewing all of the controls.

The spreadsheet area is where numeric data is displayed. It can show two dimensions at a time (indicated in the upper left corner of the matrix). The column indexes run along the top of the matrix and the row indexes are displayed along the left column. The spreadsheet area has scroll bars for viewing data elements not currently visible in the viewing area. Figure A-36, page 259, shows a close-up of the spreadsheet area.

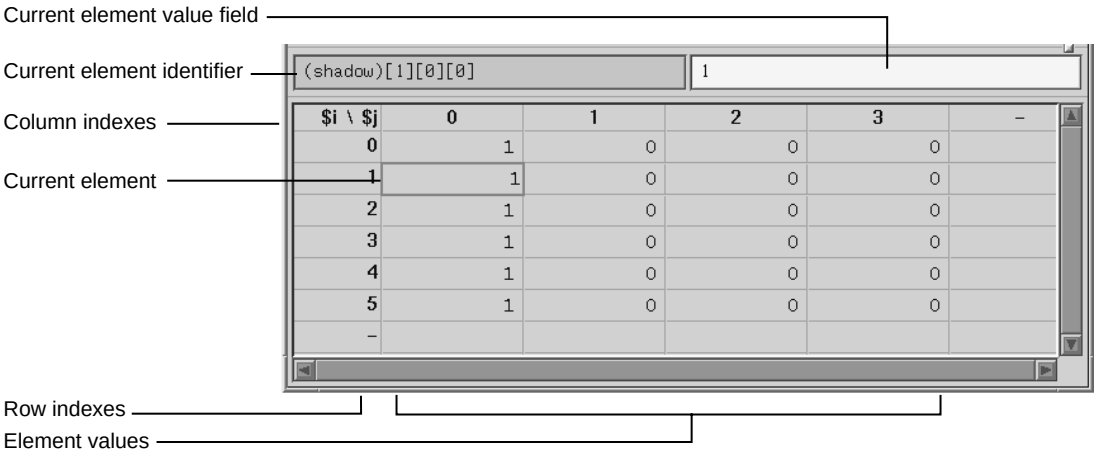

Figure A-36 Array Browser Spreadsheet Area

The current element is highlighted by a colored rectangle in the spreadsheet area. Its corresponding expression is shown in the current element identifier field, and the value is shown in the current element value field.

Spreadsheet Menu

The **Spreadsheet** menu allows you to change the appearance of data in the spreadsheet area. It provides these selections:

Column Width

Allows you to specify the width of the spreadsheet cells in terms of characters. For instance, a value of 12 indicates that 12 characters, including punctuation and digits are viewable.

Wrapped Display

Allows you to display a single dimension of an array wrapped around the entire spreadsheet area. The index value for an element is determined by adding the appropriate row index and column index values.

Figure A-37, page 260, shows an example of a wrapped array. There is only one index **$i**. The current cell is element 4 in the array (by adding 3 and +1).

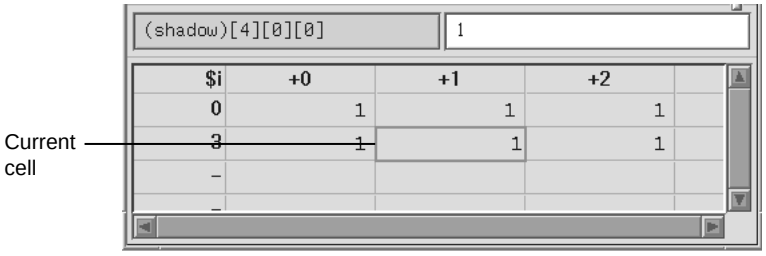

Figure A-37 Example of Wrapped Array

Auto Fit Data

If enabled, automatically resizes the spreadsheet cell to fit the maximum size of the data to be displayed. If this is enabled, then the **Column Width** is disabled. This option is on by default.

Format Menu

The **Format** menu displays a separate menu that you allows you to display the elements in the following formats:

Default toggle

Toggles the default format.

Value submenu

Contains the following display toggles for formatted values:

Decimal
Unsigned
Octal
Hex
Float
Exponential
Char
Wide Character
String

Type

Allows listing by data type.

Bit Size

Allows listing by bit size.

The graphical display area presents array data in a three-dimensional graph in one of the following formats:

- Surface (polyhedron)
- Bar chart
- Points

- Multiple lines (array vectors)

Render Menu

The **Render** tear-off menu is available only if you are have an SGI workstation with Open Inventor installed on it.

You select the graphical display mode through the **Render** menu. The **Render** menu has the following options:

Surface

Exhibits the data as a solid using the data values as vertices in a polyhedron.

Bar Chart

Presents the data values as 3-D bar charts.

Points

Simply plots the data values in 3-D space.

`Multi Line`

Plots and connects the data values in each row.

None

Allows you to display with no graphical display, in effect turning off graphical display mode.

Color Menu

The **Color** tear-off menu is available only if you are have an SGI workstation with Open Inventor installed on it.

If the **Color** menu is grayed out when the **Array Browser** window first opens, select the **Surface** option of the **Render** menu. The **Color** menu provides the following options:

Monotone Ramp

Displays the data values in a single tone, with lower numbers being darker and higher values lighter in tone.

Hue Ramp

Displays the data values in a spectrum of colors ranging from blue (lowest values) through green, yellow, orange, and red (highest values).

Exception

Allows you to flag certain conditions by color, usually for the purpose of spotting bad data. When you select **Exception**, the controls shown in Figure A-38, page 263 appear in the window.

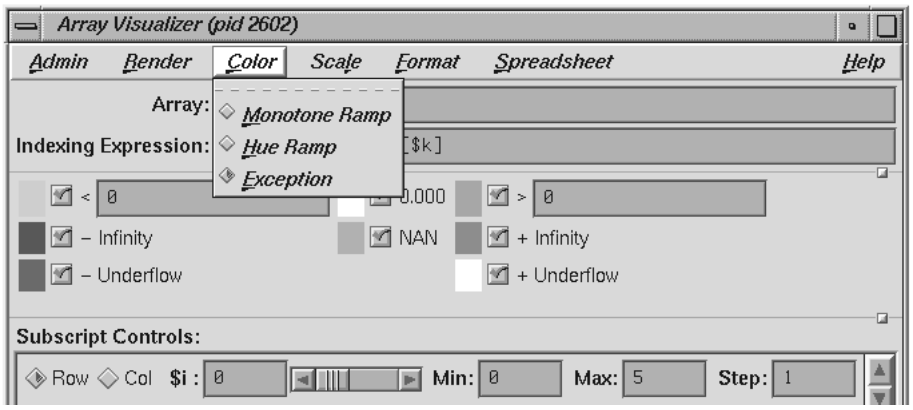

Figure A-38 Color Exception Portion of **Array Browser** Window

Thus, you can highlight data values less than or greater than specified values, values of plus or minus infinity, values of plus or minus underflow, zero values, and `NaN` (not a number) values.

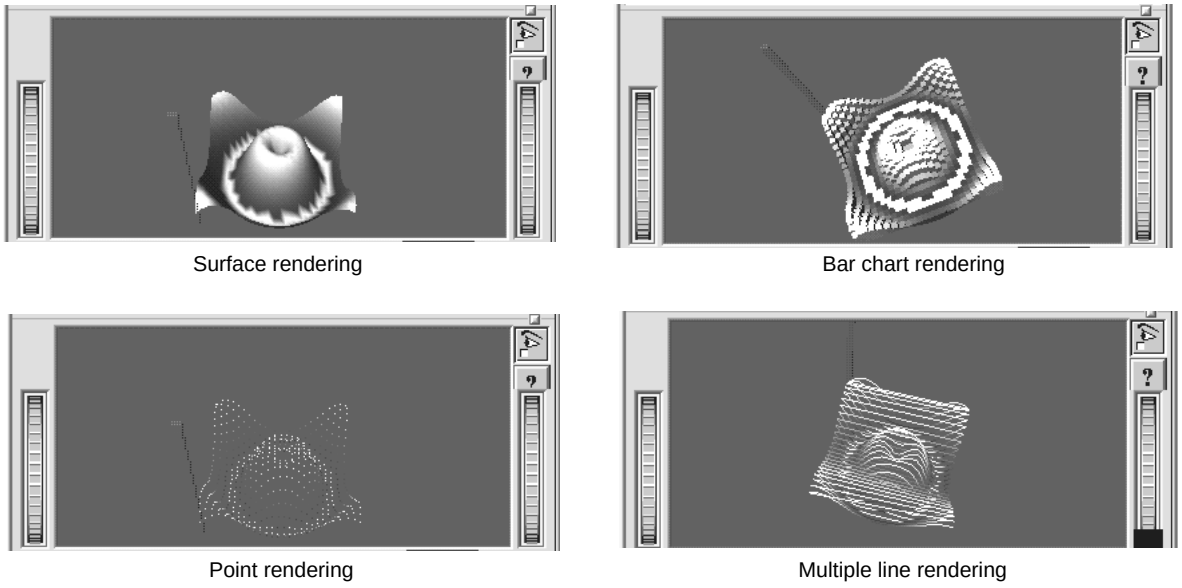

Figure A-39 Array Browser Graphic Modes

Scale Menu

Note: The **Scale** tear-off menu is available only if you are have an SGI workstation with Open Inventor installed on it.

If the **Scale** menu is grayed out when the **Array Browser** window first opens, select the **Surface** option of the **Render** menu.

The **Scale** menu provides options for changing the ratio of the *z*-dimension, which represents the value of the element. The number on the left represents the value of the *x* and *y*-dimensions (which are always the same as each other). The number on the right is the *z*-dimension.

Manipulating the *z*-dimension affects the ease of spotting differences in values. If your data is scattered over a narrow range of values, you may wish to heighten the graph by selecting **10:1** as your scale; this exaggerates the values in the *z*-dimension. If your data is in a wide range, selecting **1:2** or **1:10** as the scale will minimize the differences, flattening the graph.

Examiner Viewer Controls

Note: The Examiner Viewer is available only if you are have an SGI workstation with Open Inventor installed on it.

The graphical display uses controls and menus from Examiner Viewer. Examiner Viewer is based on a camera metaphor and borrows terms from the film industry, such as *zoom* and *dolly*, in naming its controls. The graphical display area of the window is shown in Figure A-40, page 266, with its main controls and menus. Note that the buttons on the upper right side of the graphical display area may not be visible if the area is too small; you can expose them by moving either the upper or lower sash to enlarge the display area. (The lines between the window panes include a small box to the right. Click-drag this box to change the size of the panes.)

Examiner Viewer provides these controls for viewing the graph. The right side buttons provide the following actions:

view mode

Toggles between a view-only mode (closed eye) and manipulation mode (open eye).

In view-only mode, the cursor appears as an arrow and the graph cannot be moved. Clicking on a portion of the graph selects the corresponding array element in the spreadsheet.

In manipulation mode, the cursor appears as a hand and you can move the graph. Dragging the graph with the left mouse button down moves the graph in any direction as if it were in a trackball; a quick movement spins the graph. Dragging the graph with the left mouse button and the `Ctrl` key rolls (rotates) the graph in the plane of the screen. Dragging the graph with the middle mouse button moves it without changing the viewing angle.

If you drag the graph with both the left and middle mouse buttons down, the graph will appear to move into or out of the window (this is the same as the **dolly thumbwheel**, which is described in this section).

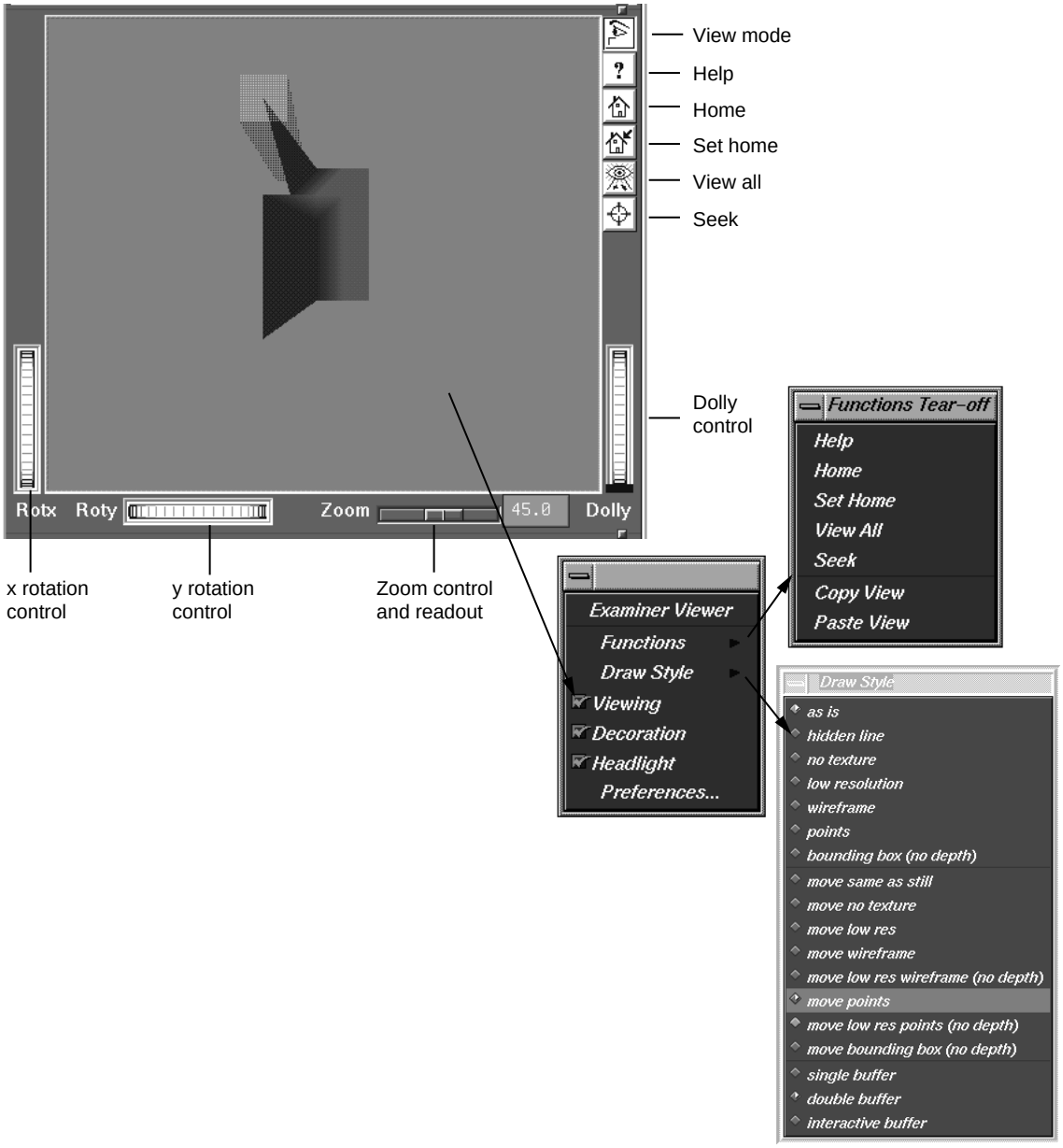

Figure A-40 Examiner Viewer with Controls and Menus

help	Runs a special help system containing Inventor Viewer information.
home	Repositions the graph in its original viewing position.
set home	Changes the home (original viewing) position for subsequent use of the **home** button.
view all	Repositions the display area so that the entire graph is visible.
seek	Provides a special cursor that allows you to reposition the graph in the center of the display area or allows you to center the view on a point you select with the cursor. See **Seek to point <or object>** in the Preferences dialog box.

The following controls let you move the graphic display:

x rotation thumbwheel

Rotates the graph around its x-axis.

y rotation thumbwheel

Rotates the graph around its y-axis.

dolly thumbwheel

Changes the size of the graph and adjusts the angles to maintain perspective. The dolly control simulates moving the viewing camera back and forth with respect to the graph.

Examiner Viewer Menu

You access the **Examiner Viewer** menu by holding down the right mouse button in the graphical display area. The **Examiner Viewer** menu provides the following options (see Figure A-40, page 266 for illustration):

Functions

Displays a submenu with the selections **Help**, **Home**, **Set Home**, **View All**, and **Seek**, which are the same as the right mouse button controls described in the previous section, and the **Copy View** and **Paste View** selections. These operate like standard copy and paste editing commands, enabling you to transfer graphs.

Draw Style

Displays a submenu that controls how the graph is displayed. The top group of options, from **as is** to **bounding box (no depth)** control how the graph is displayed when it is static. These override any **Render** menu selections.

The middle (**move...**)group of options control how the graph is displayed while in motion.

The last three options, **single**, **double**, and **interactive**, refer to buffering techniques used in moving the graph. These affect the smoothness of the movement.

Viewing

The same as the **view mode** button described in the previous section. When it is off, you can select points from the graph to display in the spreadsheet but cannot move the graph. When on, it allows you to manipulate the graph.

Decoration

Displays the right side buttons when it is on and hides them when it is off.

Headlight

Controls the shadow effect on the graph. When it is on, the light appears to come from the camera.

Preferences

Causes the **Examiner Viewer Preferences Sheet** dialog to display.

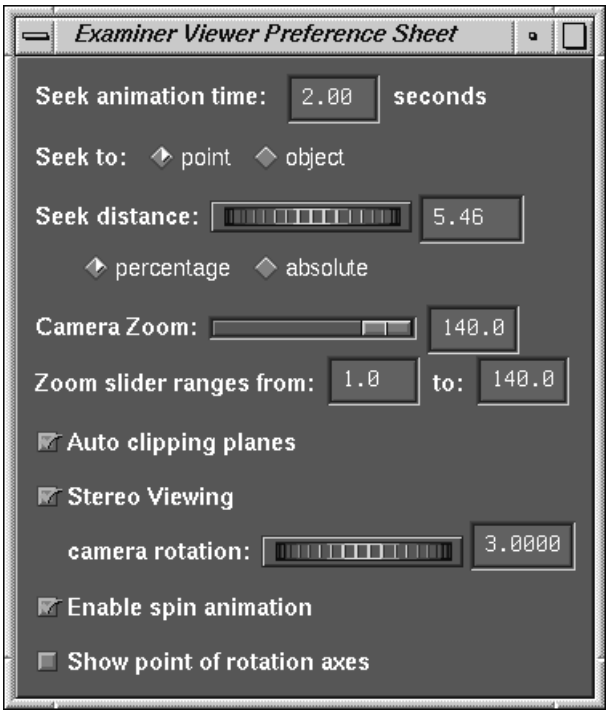

Figure A-41 Examiner Viewer Preference Sheet Dialog

The **Examiner Viewer Preference Sheet** dialog provides the following options:

Seek animation time

Allows you to specify the time it takes for the graph to be repositioned after you change the seek point. Set to `0` for instant seek. See also **Seek to point <or object>**.

Seek to point <or object>

`Seek to point` uses the picked point and surface normal to align the camera.

`Seek to object` uses only the object center to align the camera.

Seek distance wheel

This wheel controls how close to the camera the object will appear. This distance can be either an absolute distance or a percentage of the distance to the picked point.

Camera Zoom slider

This slider allows you to set (in degrees) the camera height angle (only perspective camera). If you wish, you may set this value in the field immediately to the right of the slider.

Zoom slider ranges from

Range is **1.0** to **140.0** by default, but you can reset this if you wish.

Auto clipping planes

Centers the graph in your view if enabled. If disabled, it allows you to move the graph out of view at either end of the z-axis. This is useful if you wish to focus on data above or below a set value.

Stereo Viewing

This may be turned on to see the scene in stereo (special glasses required). The offset between the left and right eye can be specified using the **camera rotation** thumbwheel.

Enable spin automation

When selected, you can cause the camera to continue spinning. To animate, left-click and drag in the direction of your desired spin, then release while spinning. The graph spins as if by a trackball. To stop, left-click anywhere.

Show point of rotation axes

Displays a set of three axes. You can move the graph around the x and y axes using the thumbwheel controls described in the previous

section. When this option is on, you can set the size of the axes in pixels.

Call Stack Window

The **Call Stack** (Figure A-42, page 271) window displays call stack entries when a process has stopped.

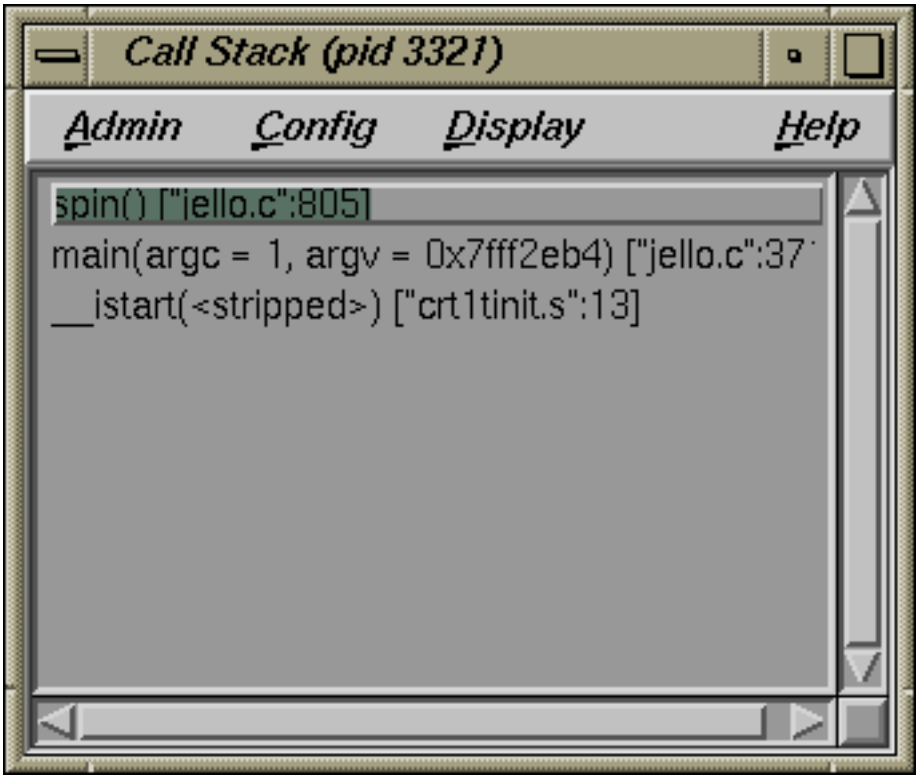

Figure A-42 Call Stack

The source display in the Main View window has two special annotations that are synchronized with the **Call Stack**:

- The location of the current program state is indicated by a large arrow representing the PC. The source line to which the arrow points is highlighted (usually in green).
- The location of the call to the item selected in the **Call Stack** window is indicated by a small arrow representing the current context. The source line becomes highlighted (usually in blue-green).

The **Call Stack** contains its own menu bar, which contains the **Admin**, **Config**, **Display**, and **Help** menus. The **Admin** menu is the same as that described in "Admin Menu", page 208. The **Help** menu is the same as that described in "Help Menu", page 196. The other menus are described in the following sections.

Config Menu

The **Config** menu contains the following option:

Preferences

Launches the **Call Stack Preferences** dialog that allows you the option of setting the maximum depth of the **Call Stack**.

Display Menu

The **Display** menu contains the following toggles which change what is displayed in each entry of the **Call Stack** in addition to the subroutine/function name:

Arg Values

Allows you to display argument values. Default is on.

Arg Names

Allows you to display argument names. Default is on.

Arg Types

Allows you to display argument types. Default is off.

Location

Allows you to display function location. Default is on.

PC

Allows you to display the program counter (PC). Default is off.

Expression View Window

The **Expression View** window is shown in Figure A-43, page 273. **Expression View** displays a collection of expressions that are evaluated each time the process stops or the context changes.

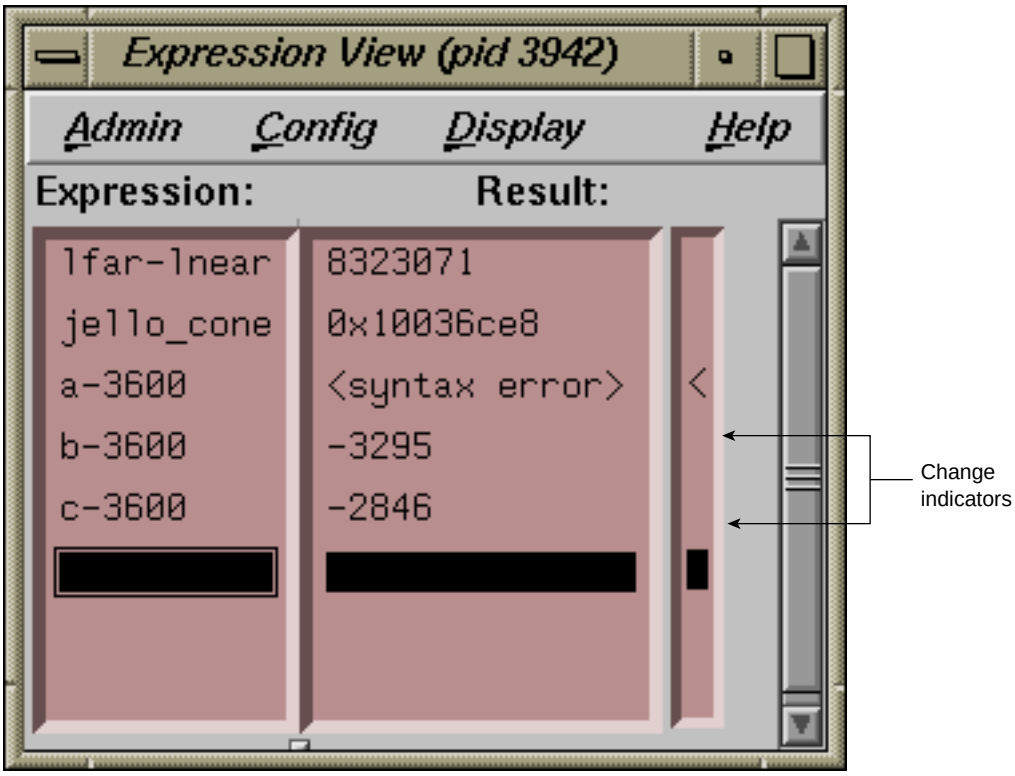

Figure A-43 Expression View

In addition to the items on the menu bar, the window has two pop-up menus: the **Language** menu and the **Format** menu. The **Admin** menu is the same as that

described in "Admin Menu", page 208. The **Help** menu is the same as that described in "Help Menu", page 196. The other menus are described in the following sections.

Config Menu

The **Config** menu contains the following options:

Load Expressions

Launches the **Load Expressions** dialog box that allows you to choose source file from which to load your expressions.

Save Expressions

Launches the **Save Expressions** dialog box that allows you to choose a file to which you can save your expressions.

Display Menu

The **Display** menu contains the following option:

Clear All

Clears all fields in the view.

Language Pop-up Menu

The **Language** pop-up menu contains three buttons that allow you to select one of three languages for evaluation: C, C++, or Fortran. The **Language** pop-up is invoked by holding down the right mouse button while the cursor is in the **Expression** column.

Format Pop-up Menu

The **Format** pop-up menu is displayed by holding down the right mouse button in the **Result** column.

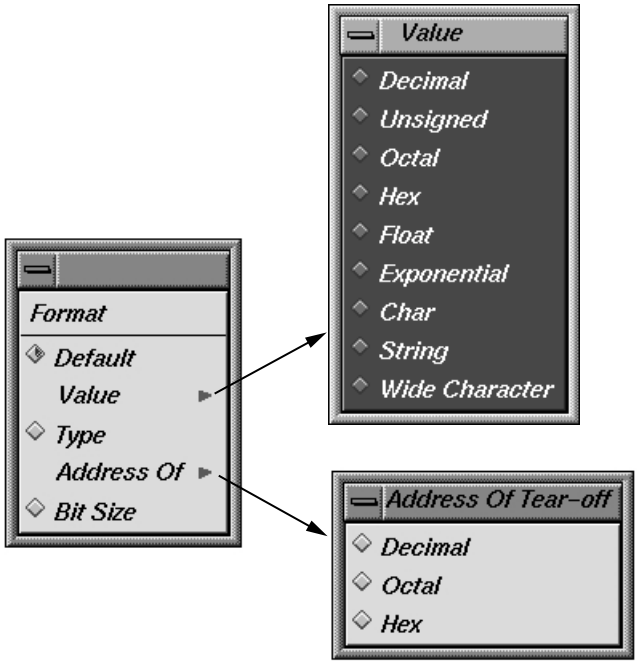

Figure A-44 Expression View Format Popup with Submenus

The **Format** popup contains the following options:

Default

Sets the format to the default values.

Value

Displays a submenu from which you can select a value of **Decimal**, **Unsigned**, **Octal**, **Hex**, **Exponential**, **Float**, **Char**, **String**, or **Wide Character** type.

Type

Displays a submenu from which you can select a type of **Decimal**, **Octal**, or **Hex**.

Bit size

Sets the format to **Bit Size**.

File Browser Window

The **File Browser** window displays a list of source files used by the current executable. Double-click on a file in the list to load it into the source display area in the Main View or **Source View** windows. Some files may be listed due to subroutines/functions being resolved from system libraries. If you select such a file, you may get the following message:

`Unable to find file <`*xxx*`.c>`

This is because the source for system library routines may not be stored on your system.

The **Search** field allows you to find files in the list.

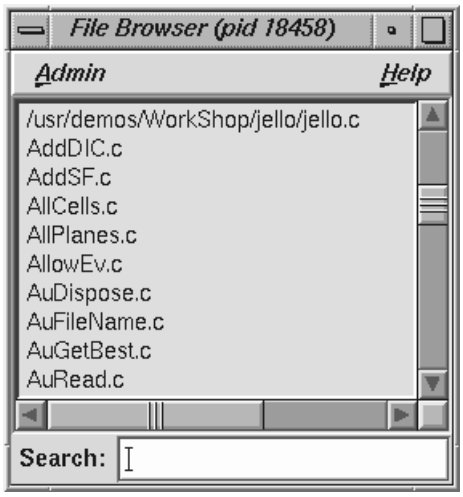

Figure A-45 File Browser Window

The **File Browser** contains an **Admin** menu (described in "Admin Menu", page 208) and a **Help** menu (described in "Help Menu", page 196). In addition, the browser has a **Search** field.

Structure Browser Window

The **Structure Browser** window allows you to examine data structures and the relationships of the data within them. It displays complex data structures as separate graphical objects, using arrows to indicate relationships. A sample **Structure Browser** is shown in Figure A-46, page 277, with the **Config**, **Display**, **Node**, and **Format** menus displayed.

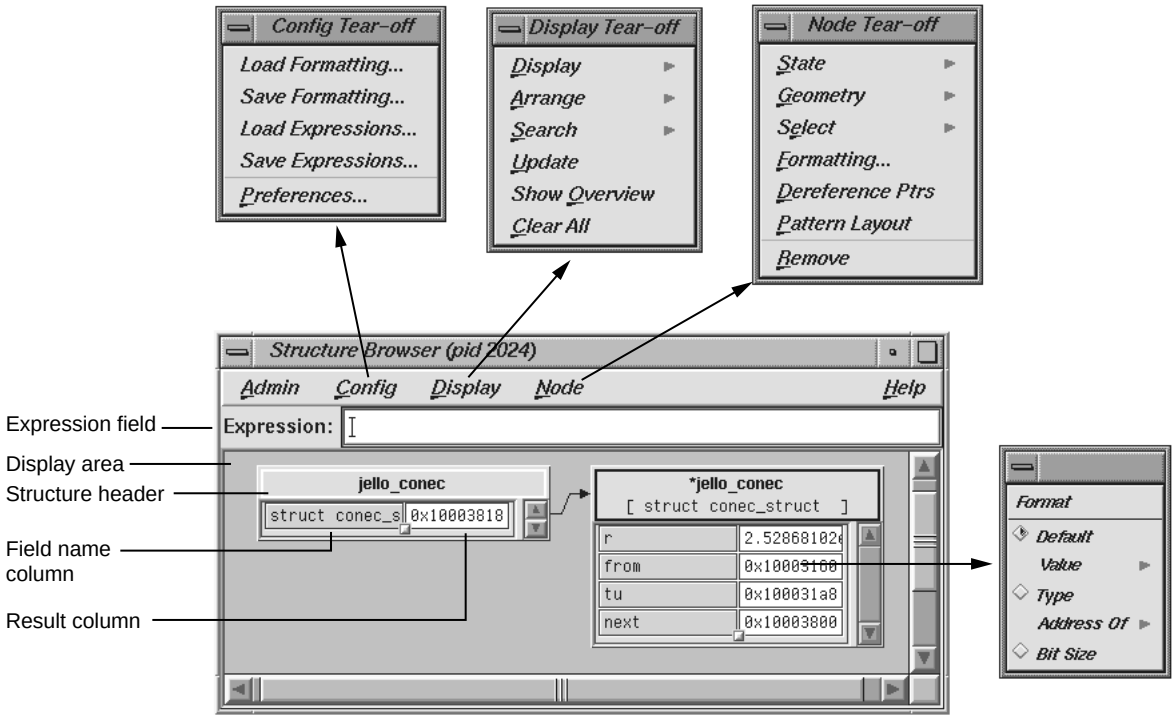

Figure A-46 Structure Browser with Menus Displayed

The structure name is entered in the **Expression** field. It then appears as an object or set of objects in the display area in the lower portion of the window. Each structure has a header identifying the structure, color coded by data type. Below the header are two columns: the left displays the field name and the right displays the field's value. If a displayed structure exceeds the size of the **Structure Browser** window, scroll bars appear.

While the **Admin** menu contains options for selecting the active structure, cloning structures, saving them as text, and closing the **Structure Browser** window; the four additional menus contain options that allow you to change the way data is displayed. The following data-display menus are available:

Config	Provides options for saving and reusing type-specific formats and expressions. You can also set preferences regarding how objects of a given type are to be displayed.
Display	Provides display options for all objects in the display area.
Node	Provides options for selected objects in the display area only.
Format	Allows you to change or reformat a specific value in the result column. To access this pop-up menu, hold down the right mouse button while the cursor is over the result column.

Using the Structure Browser Overview Window to Navigate

WorkShop provides the **Structure Browser Overview** window (from the **Show Overview** option in the **Display** menu) as another way to navigate around the display.

This window is a reduced-scale view of the requested structures. The structures are represented by solid rectangles color-coded by data type. The relative position of the currently visible area is represented by a transparent rectangle. This rectangle can be dragged (using the left mouse button) to change the display of the **Structure Browser**. Clicking the left mouse button in an area of this window repositions the currently visible area.

Entering Expressions

The **Structure Browser** accepts any valid expression. If the result type is simple, a structure displays showing the type and value. If the result type is a pointer, it is automatically de-referenced until a non-pointer type is reached. If the result type is a structure or union, an object is displayed showing the structures' fields and their values. After the expression is entered, the **Expression** field clears. The **Structure Browser** can display unrelated structures at the same time; you simply enter new structures by using the **Expression** field.

The **Expression** field is also used to enter strings used in searches.

Working in the Structure Browser Display Area

Within the display area, you select objects by clicking in the node headers. Shift-clicks add the selected object to the current selection. You can drag selected objects using the middle mouse button.

Clicking the right button while the cursor is in the right column of an object displays the **Format** menu, which is used to change the display. You can set a default format or request that results be displayed by value, type, address, or size in bits.

Holding down the right button in the header of an object brings up the **Node** pop-up menu, which is the same as the **Node** menu in the menu bar. It is used to change the way selected objects are displayed. When you left-click in the header of an object, it turns on the resizer, which allows you to change the size of the object. You will see a small square (handle) at the upper-right and lower-left corners. Left-click-drag the handle to resize the object. Middle-click–drag the handle to move the object.

Arrows show relationships among the displayed structures. If a member field is not visible in a structure, its arrow tail is displayed at the top or bottom of the scrolling area for fields. Otherwise, its tail is adjacent to its field.

Double-clicking a value field (right column) for a pointer changes the display so that the data structure it points to is displayed.

Double-clicking a member field (left column) puts the full expression for that member in the **Expression** field.

Structure Browser Display Menu

The **Display** menu controls the way structures appear in the display area. The **Display** menu provides the following options:

Display — Determines contents of the display. The **Display** option has the following two options:

- **Expression** — Displays the structure of the expression entered in the **Expression** field.
- **Selection** — Displays the structure based on the text you have selected in the source code pane in the Main View window.

Arrange — Rearranges the currently selected nodes. **Arrange** has the following two options: (See Figure A-47, page 280.)

- **Tree** — Arranges nodes into a tree-type formation, that is, the hierarchy descends from left to right and child structures are shown as branches to the right of the parent.
- **Linked List** — Arranges nodes into a linked list formation, that is, horizontally.

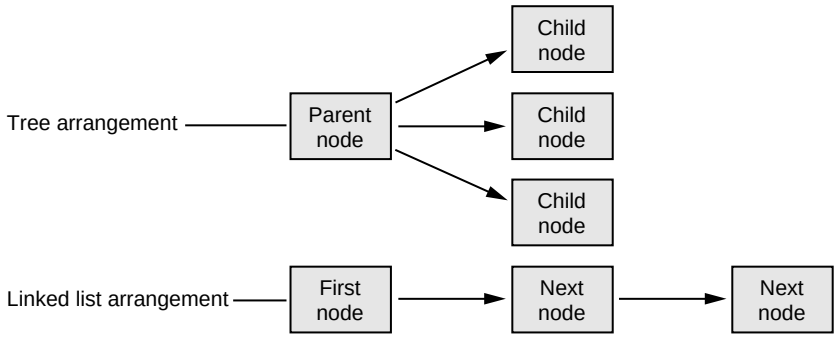

Figure A-47 Tree and Linked List Arrangements of Structures

Search

Allows you to select structures in the display area that contain the *string* specified in the **Expression** field. **Search** has the following four options:

- **Name** — Selects structures whose names contain the specified string.
- **Type** — Selects structures whose types contain the specified string.
- **Field Name** — Selects structures that have a field whose name contains the specified string.
- **Value** — Selects structures that have a field value containing the specified string.

Update

Explicitly updates the displayed structures. This happens automatically in the current **Structure Browser** when the process stops. This can be used in an inactive **Structure Browser** to update it. It can also be used to

update the display after changes have been made in other Debugger views.

Show Overview Brings up the **Structure Browser Overview** window.

Clear All Clears all structures from the display area.

Node Menu

The **Node** menu gives you options that apply to currently selected objects.

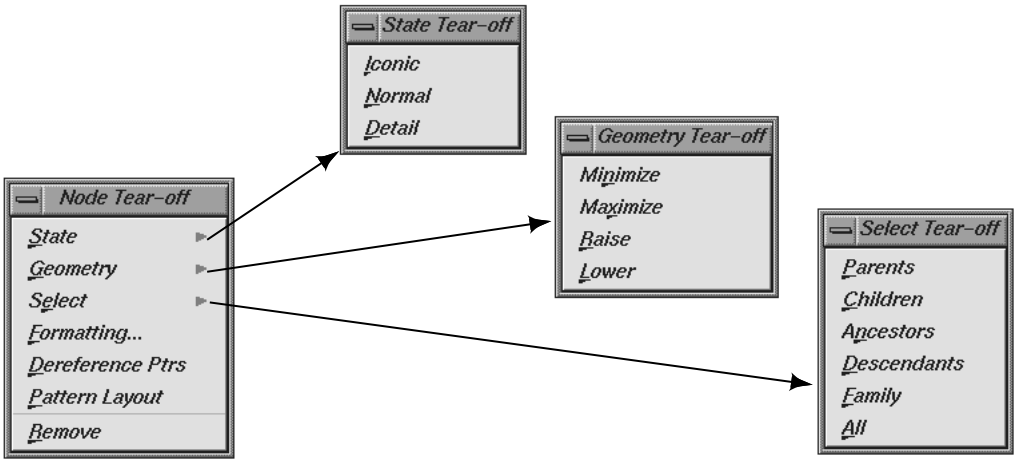

Figure A-48 Node Menu

Descriptions for these menu items are as follows:

State Controls the display of nodes. There are three options:

- **Iconic** — Displays the node header only.
- **Normal** — Uses the default chart display but hides those fields selected to be invisible.
- **Detail** — Uses the default chart display and shows all fields.

Geometry Manages graphical objects in the display area. There are four options:

- **Minimize** — Sets the vertical size of an object to the default minimum number of fields. The initial default is four fields but can be changed through either the **Formatting** selection from the **Node** menu or the **Preferences** selection from the **Config** menu.
- **Maximize** — Displays the object as large vertically as necessary to fit all of the fields.
- **Raise** — Raises the selected object(s) to the top of the display.
- **Lower** — Lowers the selected object(s) to the bottom of the display.

Select Allows you to select objects in various ways. There are six options:

- **Parents** — Selects all objects that have pointers pointing to a selected object.
- **Children** — Selects all objects pointed to by any fields in a selected object.
- **Ancestors** — Selects all objects pointed to a selected object or pointing to an object that has a descendant pointing to a selected object.
- **Descendant** — Selects all objects pointed to by any fields in a selected object or pointed to by any children of a selected object.
- **Family** — Selects all ancestors and descendants of a selected object.
- **All** — Selects all objects.

Formatting Brings up the type formatting dialog for this type.

Dereference Ptrs Dereferences any pointers in selected objects.

Pattern Layout Displays selected structures that are connected by pointers to position related structures in the same way.

Remove	Removes selected object from the display.

Formatting Fields

Each field in a data structure has certain display characteristics. These can be specified for all objects in the **Structure Browser Preferences** dialog box or for type-specific objects only in the **Structure Browser Type Formatting** dialog box. To display the **Structure Browser Preferences** dialog box, select **Preferences** from the **Config** menu (see Figure A-49).

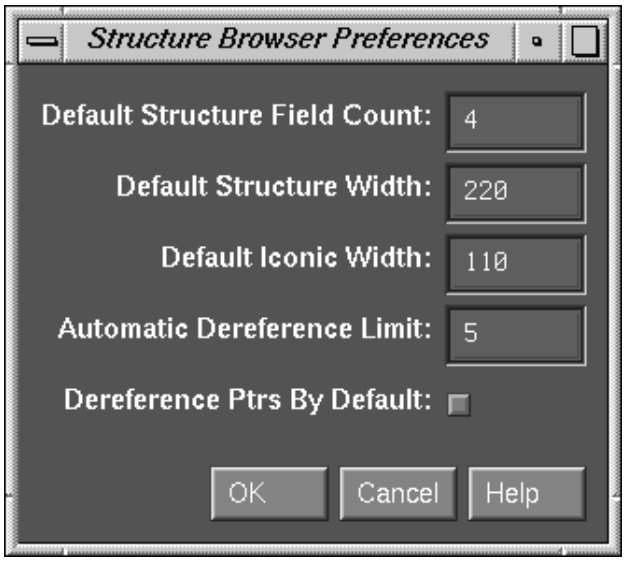

Figure A-49 Structure Browser Preferences Dialog

The dialog has the following fields:

Default Structure Field Count

Sets the number of fields to be displayed initially.

Default Structure Width

The width in pixels of the object.

Default Iconic Width

The width in pixels of the object when it is in iconic form.

Automatic Dereference Limit

Limits the number of structures that are automatically dereferenced.

Dereference Ptrs By Default

Toggles automatic dereferencing on and off.

To bring up the **Structure Browser Type Formatting** dialog box, select the set of structures under consideration and select **Node Formatting** from the **Node** menu (see Figure A-50, page 285).

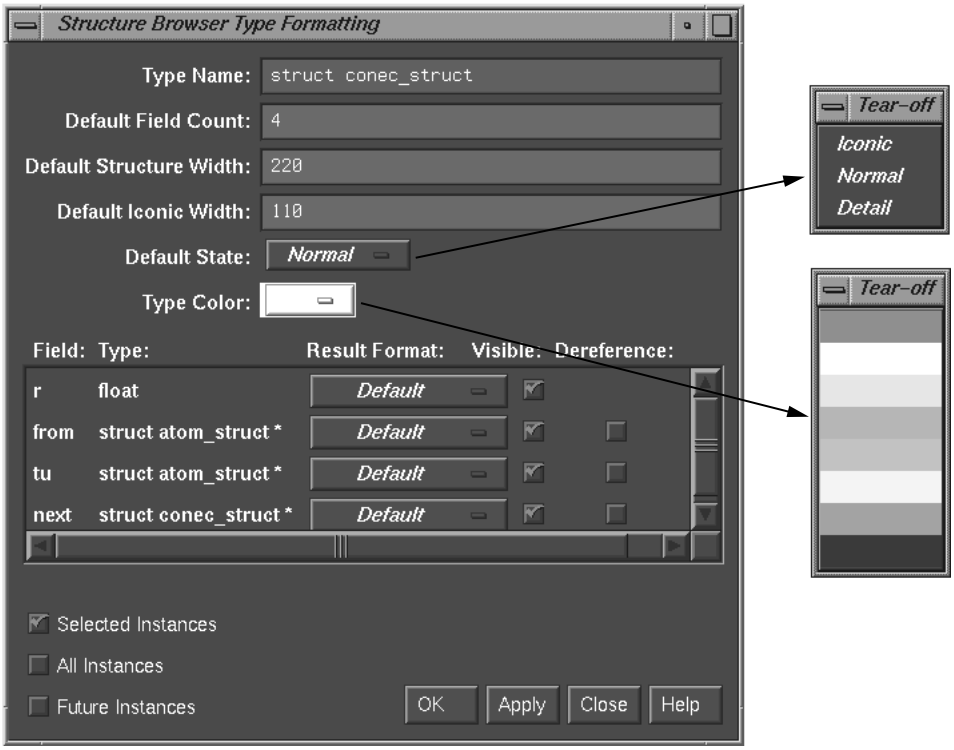

Figure A-50 Structure Browser Type Formatting Dialog

The dialog box has the following fields:

Type Name

Displays the current data type.

Default Field Count

Lists number of fields to be displayed initially for objects of that type.

Default Structure Width

Displays the width in pixels of the object.

Default Iconic Width

Displays the width in pixels of the object when it is in iconic form.

Default State

Brings up a pop-up menu that allows you to specify whether structures are first displayed as icons (**Iconic**), with the minimum number of fields displayed (**Normal**) or with all fields displayed (**Detail**).

Type Color

Provides a submenu for color coding. It allows you to select a color for the header and overview rectangles for objects of a given type.

For structure and union types, the list box shows all the fields with their types. For each field, you can change the result format to one of the following types:

- Default
- Decimal
- Unsigned
- Octal
- Hex
- Float
- Exponential
- Char

- String
- Wide Character
- Type
- Dec addr
- Oct addr
- Hex addr
- Bit Size

You can also specify whether a field is visible in normal state, and if it is a pointer field, whether it should be automatically dereferenced.

Once you specify the format for this type, you can apply it to any combination of the following through the toggle buttons in the bottom left portion of the window:

- Selected instances
- All existing instances
- Any future instances of this type

Variable Browser Window

The **Variable Browser** window allows you to view and change the values of local variables and arguments at a specific point in a process. (Global variables can be viewed or changed using **Expression View** or the **Evaluate Expression** selection from the **Data** menu for one-shot evaluations.) In addition to providing values, the **Variable Browser** is useful for getting a quick list of the local variables in a scope without having to search for their names. A sample **Variable Browser** window with the **Language** and **Format** menus displayed is shown in Figure A-51, page 288.

Typically, you inspect variable values at the following points:

- At a breakpoint
- At a frame in a call stack
- As you step through a process

A useful technique is to set a trap at the entry to a function and inspect the values of the variables there. Some variables may be in an uninitialized state at that point. You

can then step through the function and make sure that no uninitialized variables are used inadvertently.

Note: In programs compiled by using the `-n32 -g` compiler options, you cannot click on or print an unused variable as you could when using the `-o32 -g` options.

Entering Variable Values

The **Variable Browser** allows you to change the values of variables in the window. You simply enter the new value in the result column and press `Enter`. Thus, you can force new values into the process and see their effect.

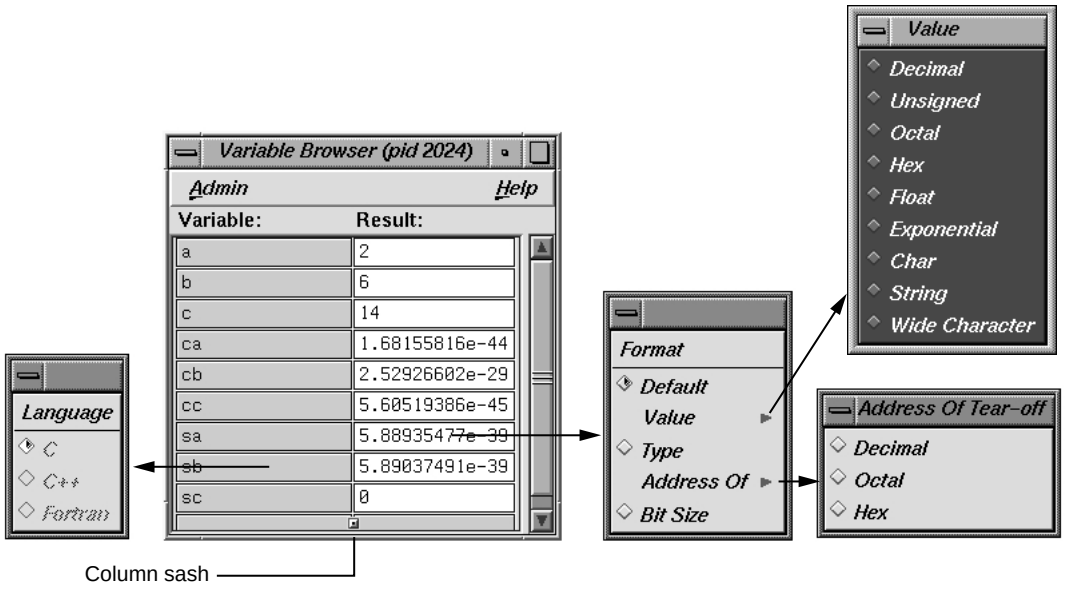

Figure A-51 Variable Browser with Menus Displayed

Changing Variable Column Widths

By using the sash between the columns, you can adjust the relative widths of the **Variable** and **Result** columns (see Figure A-51, page 288). For example, you may wish to adjust for short variable names and long result values.

Viewing Variable Changes

The Debugger views that are involved with variables (that is, the **Variable Browser** and **Expression View**) have indicators that show when the variable has changed since the last breakpoint. If you click the indicator, you can view the previous value. The variable change indicators for a **Variable Browser** window are shown in Figure A-52, page 289.

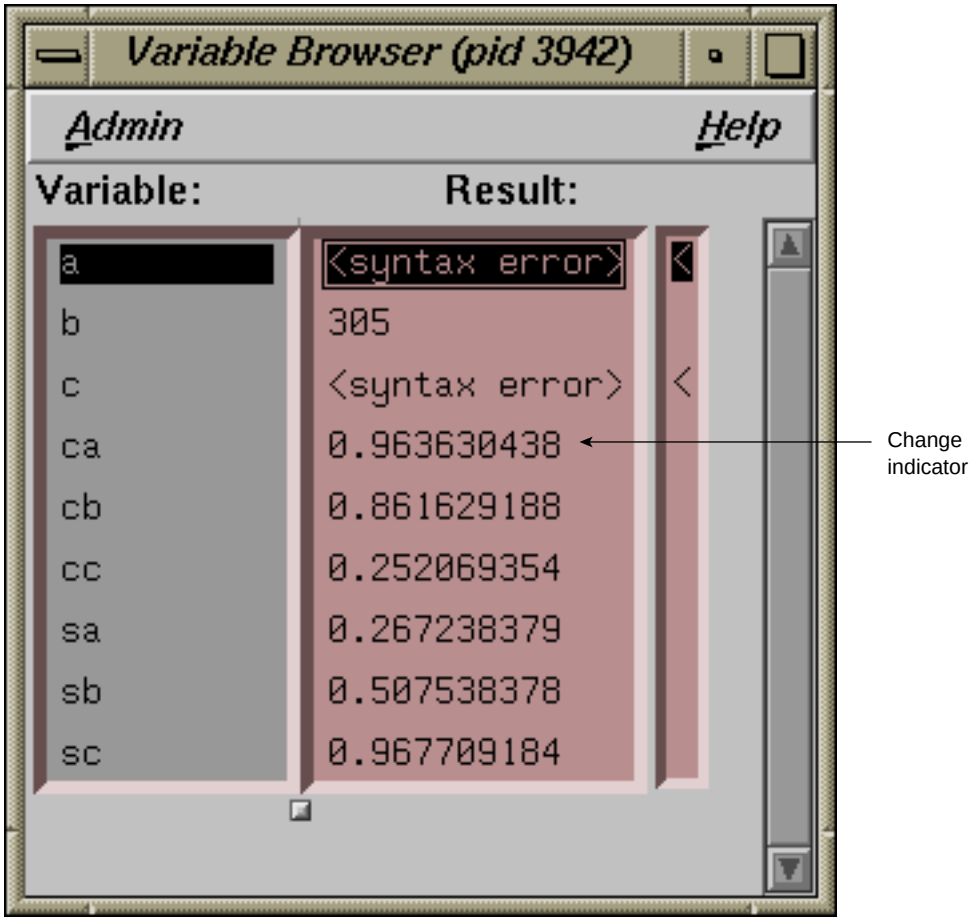

Figure A-52 Variable Browser Change Indicator

Machine-level Debugging Windows

The Debugger offers three views useful in debugging at the machine level: the **Disassembly View**, **Register View**, and **Memory View**.

The Disassembly View Window

The **Disassembly View** window allows you to look at machine-level code rather than source-level code. A typical **Disassembly View** window is brought by selecting the following from the Main View window menu bar:

Views
    Disassembly View

The window is shown in Figure A-53, page 291, with the **Disassemble** menu displayed.

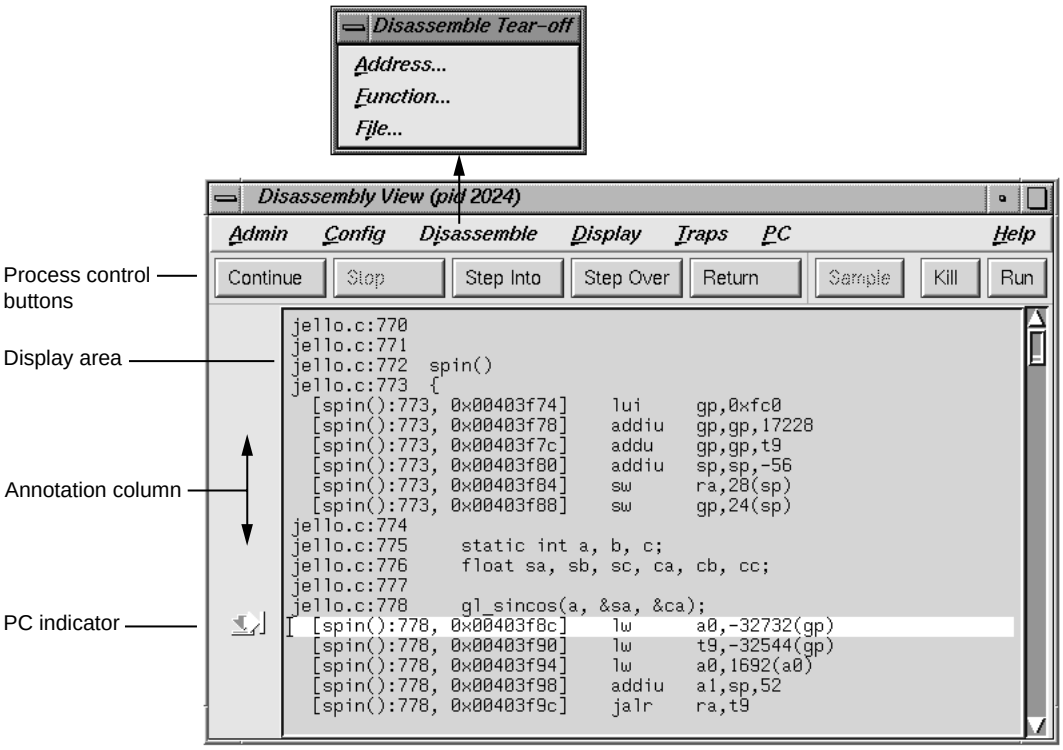

Figure A-53 The **Disassembly View** Window with the Disassembly Menu Displayed

Similarities with Main View Window

At the top of the window are the same process control buttons as those in the Main View window. They behave the same way except for **Step Into** and **Step Over**, which do machine-level instruction stepping instead of source-level. Remember that you can select the number of steps by holding down the right mouse button over the **Step Into** and **Step Over** buttons.

The **Admin**, **Display**, **Traps**, and **PC** items on the menu bar basically work same as their counterparts in the Main View window. The exception is that the **PC** submenu selections **Continue To** and **Jump To** are based on machine-level instructions rather than source-level steps.

The **Disassemble** menu is discussed in "The Disassemble Menu", page 292. The **Preferences** selection from the **Config** menu is discussed in "The Config Menu Preferences Dialog", page 294.

You can set traps either by using the **Traps** menu or by clicking in the annotation column to the left of the source display area that contains the disassembled code.

The Disassemble Menu

The **Disassemble** menu allows you to display disassembled code. It contains the following items:

Address

Allows you to disassemble a specified number of lines, starting from a specified source line address (see Figure A-54).

Figure A-54 The **Disassemble From Address** Dialog

Function

Allows you to disassemble a specified number of lines, starting from the beginning address of a specified function name (see Figure A-55, page 293).

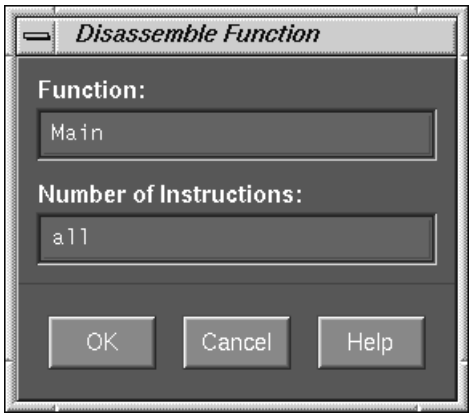

Figure A-55 The **Disassemble Function** Dialog

File

Allows you to disassemble a specified number of lines, starting from the address corresponding to a specified line number in a specified file (refer to Figure A-56, page 294). If you have a current selection in the Main View window or the **Source View** window, its file and cursor position are used as the default file name and line number, respectively.

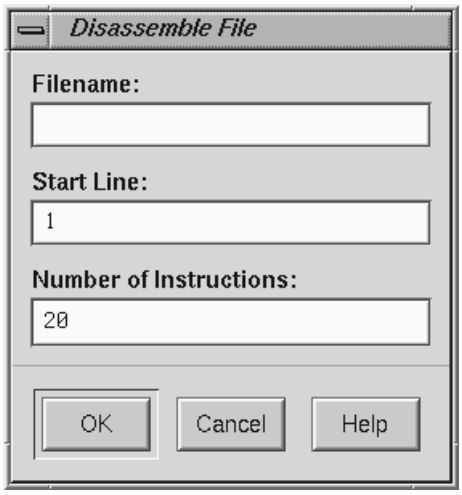

Figure A-56 The **Disassemble File** Dialog

The Config Menu Preferences Dialog

Selecting **Preferences** from the **Config** menu brings up the **Disassembly View Preferences** dialog box (shown in Figure A-57, page 295) so that you can select the display preferences you desire.

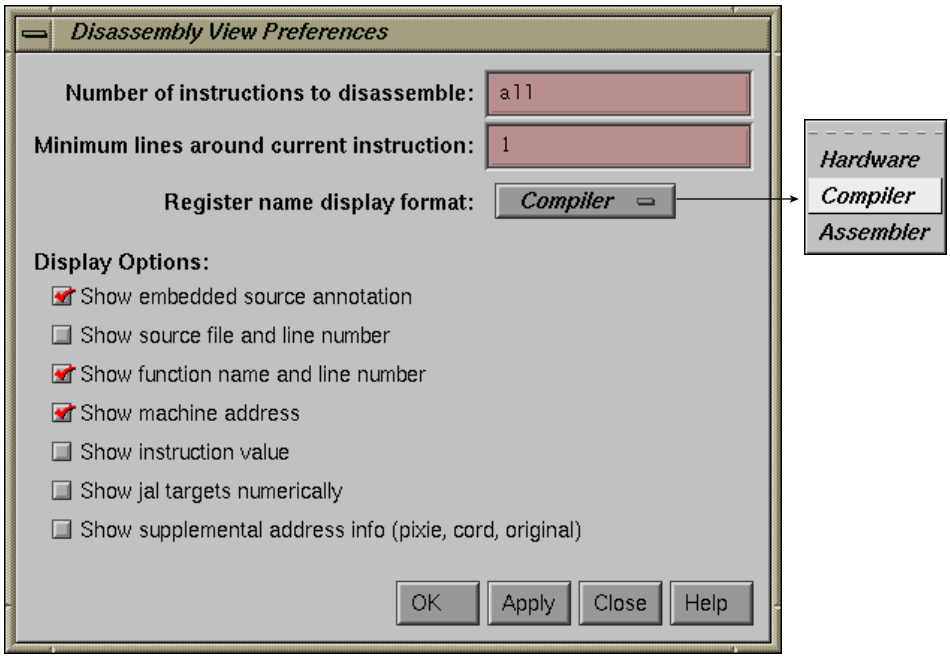

Figure A-57 The **Disassembly View Preferences** Dialog with Display Format Menu

The dialog box provides you with the following options:

Number of instructions to disassemble

Controls the default number of disassembly lines shown when the process stops. This number appears in the dialog boxes selected from the **Disassemble** menu (see Figure A-54, page 292, Figure A-55, page 293, and Figure A-56, page 294). The default is all instructions, indicating that the entire function will be disassembled.

Minimum lines around current instruction

Controls the display of the disassembled code, enabling you to view at least the specified number of instructions before and after the current instruction.

Register name display format

Controls how register names are displayed. The available modes are **Hardware**, **Compiler**, and **Assembler**.

Show embedded source annotation

When **ON**, displays source and disassembly statements interleaved. When **OFF**, displays disassembly statements only.

Show source file and line number

Displays the filename and file position along with each machine instruction.

Show function name and line number

Displays the function name and file position along with each machine instruction.

Show machine address

Displays the memory address of each machine instruction.

Show instruction value

Displays the instruction word along with each machine instruction.

Show jal targets numerically

Controls whether the target address of a `jal` instruction is displayed as a hex address or symbolic label.

Show supplemental address info (pixie, cord, original)

Displays additional address information. This may be used to set address breakpoints in the Main View command window for corded or pixified code.

The Register View Window

Register View window allows you to examine and modify register values. You bring it up by selecting **Register View** from the **Views** menu in the Main View window. Figure A-58, page 297, shows a typical **Register View** window that has been resized to show all available registers.

The **Register View** window displays each register with its current value. A question mark (?) displayed immediately before a register value signifies that the value is suspect; it may not be valid for the current frame. This can occur if a register is not saved across a function call. A colored marker indicates that a register value has changed since the last time the process stopped.

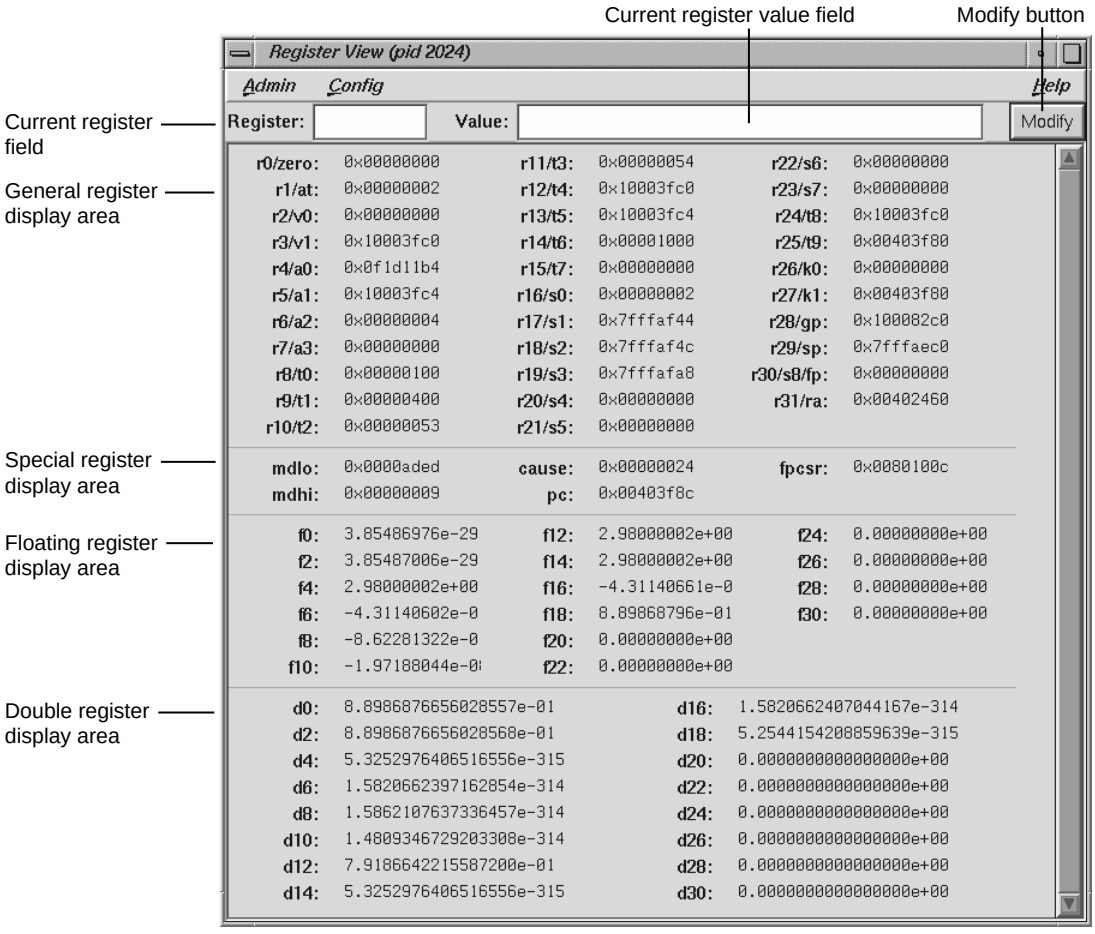

Figure A-58 The **Register View** Window

The Register View Window

The major features of the **Register View** window are the following:

`Current register` field

Identifies the currently selected register. You can switch to a different register by entering its name (either by hardware name or by alias) in this field and pressing `Enter`. You can also switch registers by clicking on the new register in the display area.

`Current register value` field

Shows the contents of the selected register. You can assign a new value to a register by entering either a literal or an expression into the **Value** field. You then click on the **Modify** button to change the value or press `Enter`.

`Register display area`

Shows the registers organized into four groups: general, special, floating, and double. Note that the general registers are identified by both their hardware and software names. For systems with 32–bit processors (O2s, for example), double precision registers represent a pair of floating-point registers. For systems with 64–bit processors (Origin2000s, for example), float registers are not displayed at all. Floating-point calculations are done in double precision registers.

Note: The special registers **p0**, `p1`, and **p2** do not appear in Figure A-58, page 297. These are used for instrumentation and appear only when instrumentation has taken place.

Changing the Register View Display

The **Preferences** selection in the **Config** menu allows you to change the **Register View** display. It brings up the **Register View Preferences** dialog box (see Figure A-59, page 299).

The register display toggle buttons let you specify which types of registers are to be displayed by default.

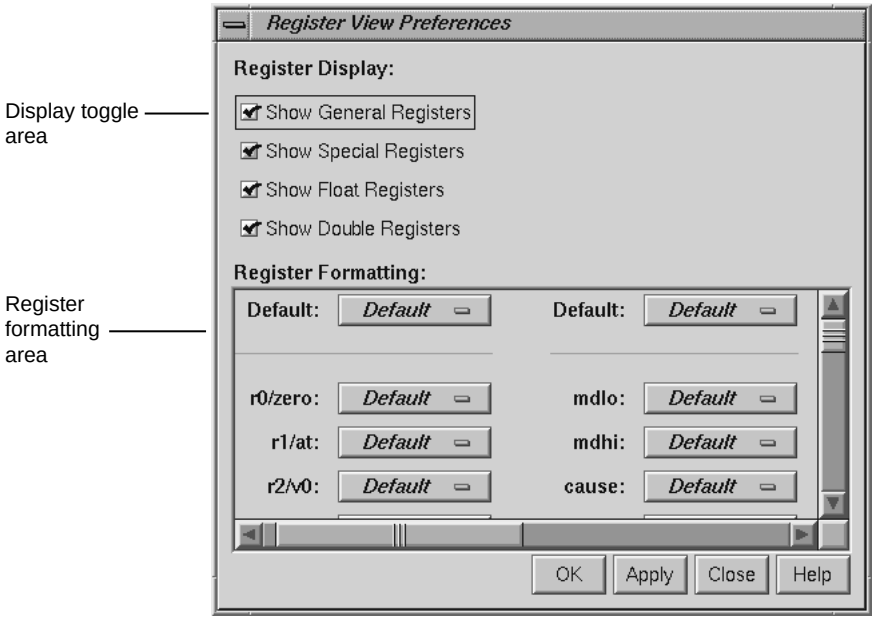

Figure A-59 The **Register View Preferences** Dialog

The register formatting area allows you to select formats for any of the registers.

The default fields in the top row let you change defaults for the four major types, which are set as follows:

- General registers — hexadecimal
- Special registers — hexadecimal
- Float registers — floating point
- Double registers — floating point

The rows in the register formatting area let you change the modes for the individual registers.

The Memory View Window

The **Memory View** window allows you to examine and modify memory. A typical **Memory View** window appears in Figure A-60, page 300.

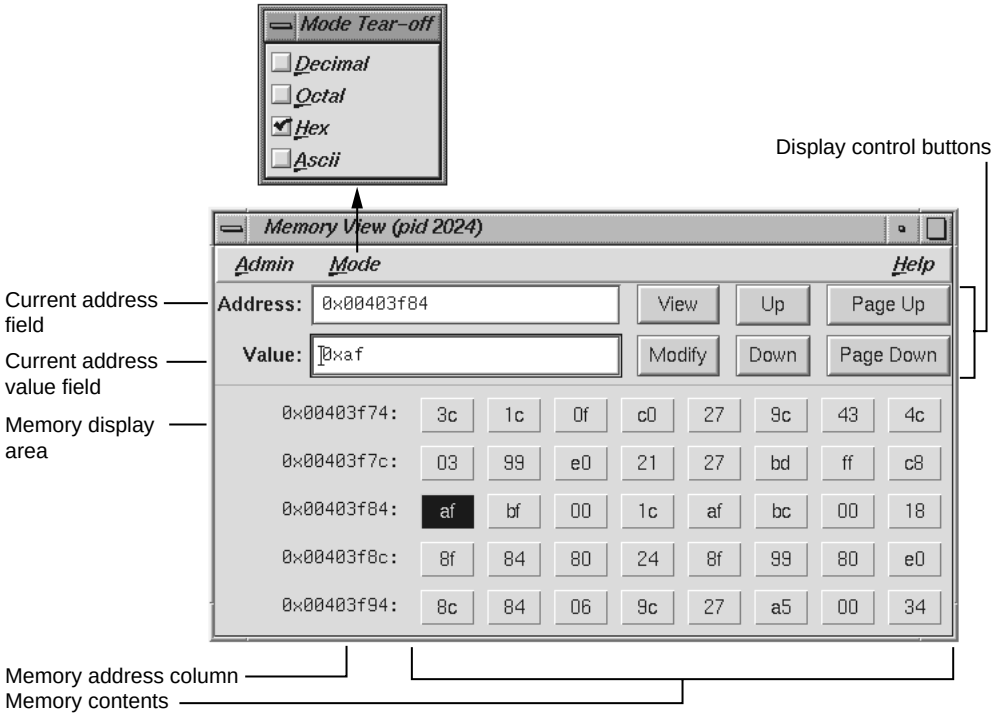

Figure A-60 The **Memory View** Window with the **Mode** Submenu Displayed

Viewing a Portion of Memory

To view a portion of memory, enter the beginning memory location in the **Address** field. You can enter the literal value or an expression that evaluates to an integer address. These address specifications must be in the language of the current process as indicated by the call stack frame. The syntax of this expression depends upon the language of the program. For example, you can enter **`0x7fff4000+4`** as the memory address when stopped in a C function or enter **`$7fff4000+4`** as the equivalent for a Fortran routine. Press `Enter` while the cursor is in the field or click the **View** button to display the contents of that location and the subsequent locations in the display

area. This also displays the contents of the first address in the **Value** field where it can be modified.

The memory display area shows the contents of individual byte addresses. The column at the left of the display shows the first address in the row. The contents at that address are shown immediately to its right, followed by the contents of the next seven byte locations. If you enlarge the **Memory View** window, you can see additional rows of memory.

Changing the Contents of a Memory Location

To change the contents of a memory location, you select the address to be changed, either by direct entry or by clicking on the byte value in the display area. You can enter a single value or a sequence of hex byte values separated by spaces (for example, **`00 3a 07 b2`**) in the **Value** field. You can also enter a quoted string to change a consecutive range of values to the ASCII values of that string. Pressing `Enter` while the cursor is in the **Value** field or clicking the **Modify** button substitutes the new value(s) starting at the specified location.

Changing the Memory Display Format

The **Mode** menu allows you to change the format of the value field and byte locations to either decimal, octal, hex, or ASCII.

Moving around the Memory View Display Area

The four control buttons at the upper right of the window help you move around the display area. These buttons are:

Up

Moves displayed bytes up a single row.

Down

Moves displayed bytes down a single row.

Page Up

Moves displayed bytes upward by as many rows as are currently displayed.

Page Down

Moves displayed bytes downward by as many rows as are currently displayed.

Fix+Continue Windows

The Fix+Continue utility interacts with several Debugger windows. The Main View and **Source View** windows access the Fix+Continue utility from the menu bars. The standard output results of running redefined code are displayed in the **Execution View** window (refer to "Changes to Debugger Views", page 310).

Special line numbers (in decimal notation) applied to redefined functions appear in several views.

Note: Fix+Continue functionality within the debugger is limited to C++ programs compiled with the `-o32` compiler option.

The following windows, devoted entirely to Fix+Continue functions, can be brought up by selecting **Fix+Continue > View** from the Main View window menu bar:

- **Status Window**
- **Message Window**
- **Build Environment Window**

This section describes Fix+Continue menu selections and windows.

The **Fix+Continue** menu is available from the Main View menu bar. The menu selections operate on the selected code or the file shown in the **Source View** window. The **Fix+Continue** menu is also available from **Source View** window and from the **Fix+Continue Status** window.

Fix+Continue Status Window

The **Fix+Continue Status** window (see Figure A-61, page 303) provides you with a summary of the modifications you have made during your session. It also allows you quick access to your modifications and somewhat expanded menu options.

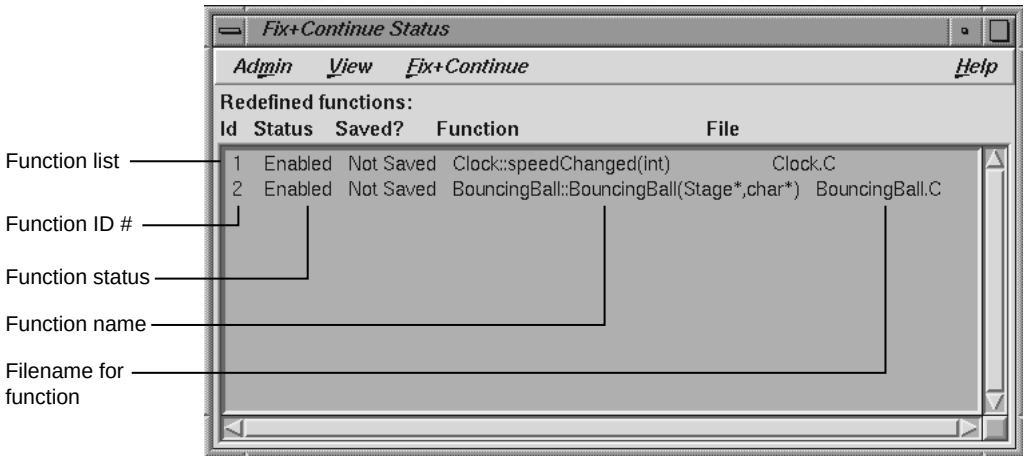

Figure A-61 Fix+Continue Status Window

The function ID number, status, name, and file name are displayed in the window. Double-clicking a line in the window brings up the corresponding source in the Debugger main window.

Fix+Continue Status window menus and submenus are described in Figure A-62, page 304.

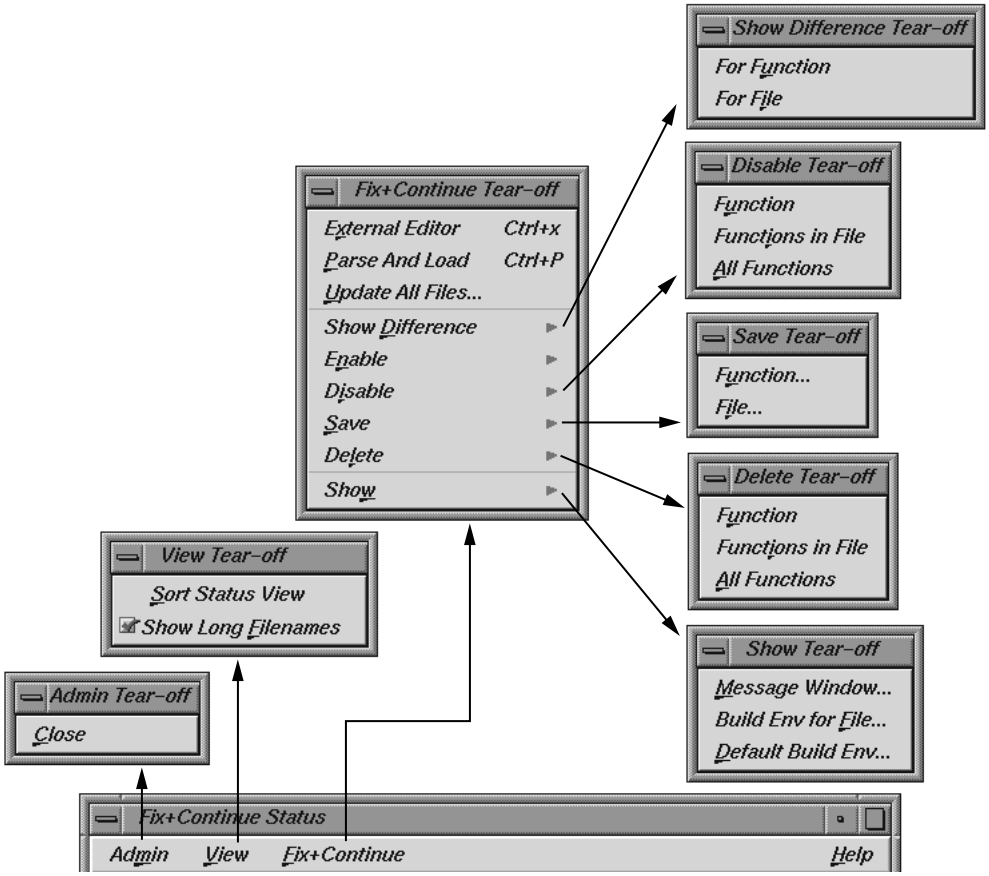

Figure A-62 Fix+Continue Status Window Menus

Admin Menu

The **Admin** menu contains an option for closing the window.

Close

Closes the status window.

View Menu

The **View** menu contains options for sorting information in the window and displaying file names.

Sort Status View

Sorts the information in the status view according to the field currently selected.

Show Long Filenames

Toggles among absolute (long) path names or base names.

Fix+Continue Menu

The **Fix+Continue** menu available from the **Fix+Continue Status** window is somewhat different from that available through the Debugger Main View. It contains a number of options and submenus that are described below. These options and submenus are active on the item you select in the source pane of the Main View window. Before using this menu, you should select an item by clicking on it. The following options and submenus are available:

External Editor

Allows you to edit with an external editor such as `vi`, rather than the Debugger's default editor.

Parse And Load

Parses your modified code and loads it for execution. You can execute the modified code by clicking on the **Run** or **Continue** command buttons in the Main View window.

Update All Files

Launches the **Save File+Fixes As** dialog that allows you to update the current session while saving all the modifications to the appropriate files.

Show Difference submenu

Allows you to show the difference between the original source and your modified code. You can show the difference in the code in one of the two following ways:

- **For Function** — Shows the differences for the current function only.
- **For File** — Shows the differences for the entire file that contains the current function.

Enable submenu

Allows you to enable the changes in your modified code in one of the following ways:

- **Function** — Enables the changes in the current function.
- **Functions in File** — Enables the changes to the current function in its own file.
- **All Functions** — Enables the changes to all functions in the modified code.

Disable submenu

Has the same menu choices as the **Enable** submenu, but disables rather than enables.

Save submenu

Allows you to save your code changes to a file. You can save the changes in one of the following ways:

- **Function** — Launches the File dialog, allowing you to save only the current function to a file.
- **File** — Launches the **Save File+Fixes As** pop-up window allowing you to save the entire file that contains the current function.

Delete submenu

Has the same menu choices as the **Save** submenu, but deletes rather than saves.

Show submenu

Allows you to launch any of the following windows:

- **Message Window** — Launches a **Fix+Continue Error Messages** window for the selected item. See "Fix+Continue Error Messages Window", page 307, for more details.
- **Build Env for File** — Launches a **Fix+Continue Build Environment** window for the file shown in the **Source View** window. See "Fix+Continue Build Environment Window", page 308, for more details on the **Fix+Continue Build Environment** window.
- **Default Build Env** — Launches the **Fix+Continue Build Environment** window to show the options that are to be used in cases where they could not be obtained from the target. See "Fix+Continue Build Environment Window", page 308, for details on the **Fix+Continue Build Environment** window.

Fix+Continue Error Messages Window

The **Fix+Continue Error Messages** window contains a list of errors and other system messages that pertain to your source modifications, parses, and attempts to run your modified source.

You can highlight the source line where the error occurred by double-clicking the appropriate line in the window. The **Fix+Continue Error Messages** window contains the following buttons:

Clear

Clears all the parsing errors and warnings.

Next

Puts a check mark on the next unchecked error warning entry in the parse messages. It displays the corresponding file and line in the **Source** view, highlighting it according to the type of error or warning. The **Next** option does not work after all the entries in the messages are ticked.

Rescan

Erases all the ticks, so that you can rescan all the error warnings from the beginning.

Admin Menu

The **Admin** menu allows you to perform either of the following operations:

Clear All

Clears all messages in the window.

Close

Closes the window.

View Menu

The **View** menu allows you to set any of the following toggles:

Show Warnings

Causes compile warnings to be displayed in the parse errors list.

Append Parse Messages

Causes parse messages to be appended to the parse errors list.

Append Load Messages

Causes load messages to be appended to the load errors list.

Fix+Continue Build Environment Window

This section describes the **Fix+Continue Build Environment** window (see Figure A-63, page 309). The **Fix+Continue Build Environment** window provides you with the build information for your source code in your current environment. It displays the command that was used to build your executable and the name of the file that contains the function that you currently have selected.

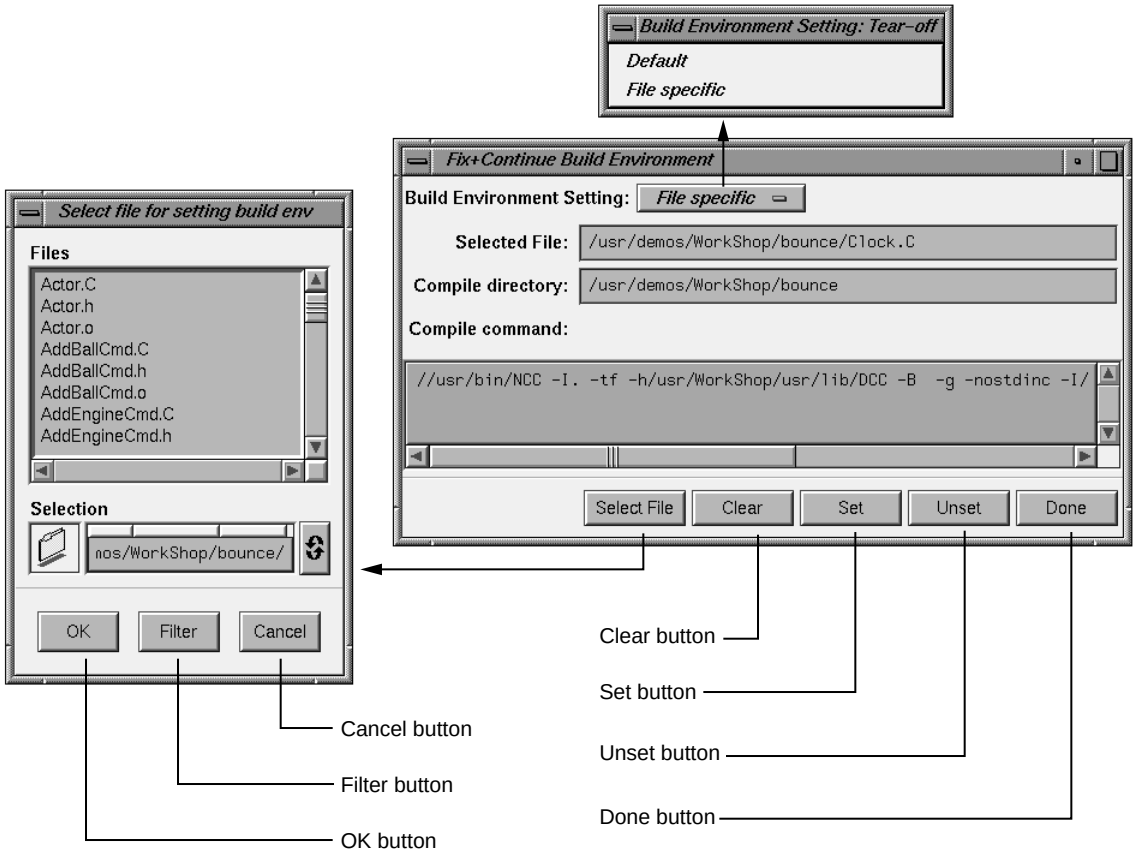

Figure A-63 Fix+Continue Build Environment Window

The compiler and associated flags that were used to compile the file are normally gathered from the target. You can use this window to make any changes to these flags.

The window allows you to select your build environment setting through the **Build Environment Setting** toggle that contains the following two options:

Default — Sets the build environment to default that is displayed in the window.

File Specific — Sets the build environment to that of the file that contains the currently selected function. You can

change the file by clicking the **Select File** button, which launches the **File** dialog.

The **Fix+Continue Build Environment** window also contains the following buttons:

Select File	Launches the **File** dialog and allows you to select a file from which to set the build environment.
Clear	Clears the window.
Set	Sets the build environment to what is displayed in the window.
Unset	Unsets the build environment.
Done	Dismisses the window.

Changes to Debugger Views

When you use Fix+Continue, views change to show redefined functions or stopped lines containing redefined functions.

Main View

All Fix+Continue actions are available through the **Fix+Continue** menu on the Main View window. See Figure A-64, page 311, for details.

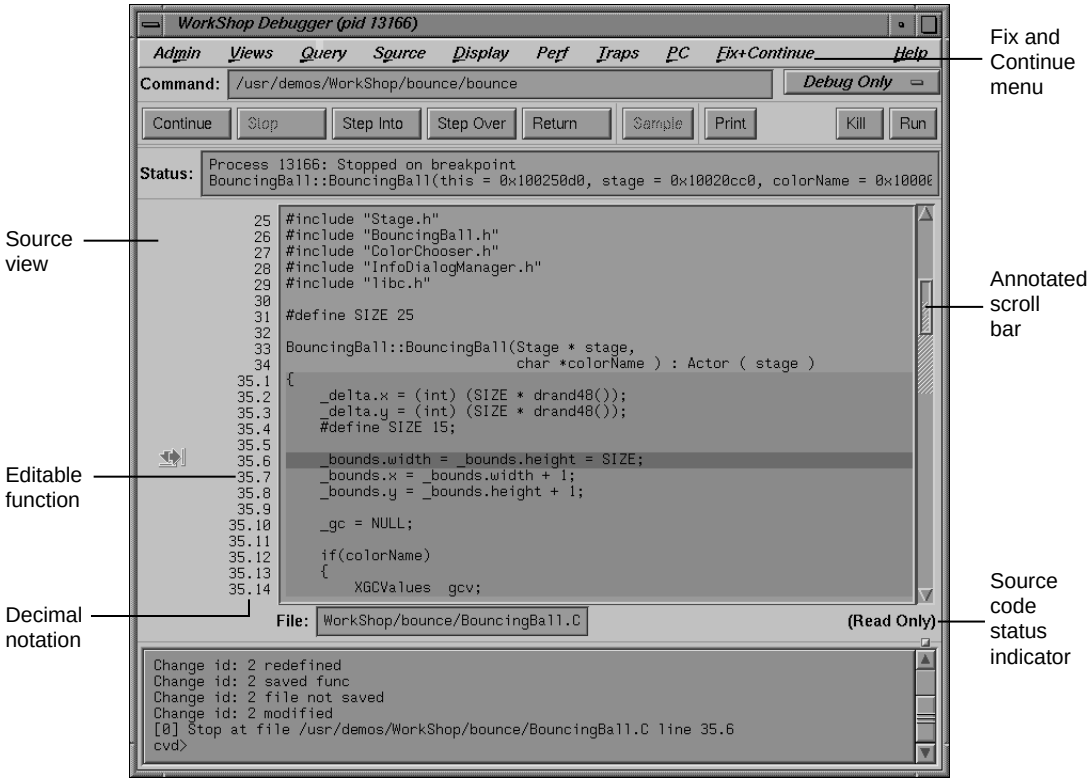

Figure A-64 Debugger Main View Window

You can select commands from the **Fix+Continue** menu or enter them at the Debugger command line. The source code status is `Read Only`. Color coding shows the differences between editable code, enabled redefinitions, disabled definitions, and breakpoints. Line numbers in redefined functions have decimal notation that is used for every reference to the line number. The integer portion of the decimal is the same as the first line of the function. This ensures that compiled source code line numbers remain unchanged.

Command Line Interface

The Debugger command line interface accepts Fix+Continue commands and reports status involving redefined functions or files. Figure A-65, page 312, shows a function

successfully redefined using the command line. Change id 1 was previously redefined and assigned the number 1.

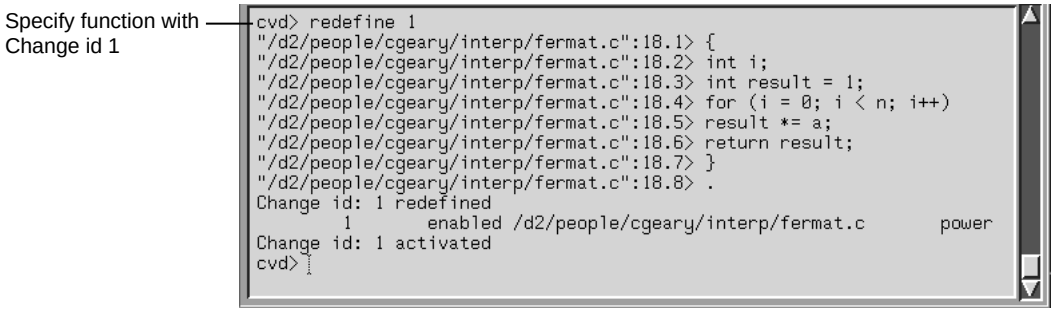

Figure A-65 Command Line Interface with Redefined Function

Call Stack

The **Call Stack** recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure A-66.

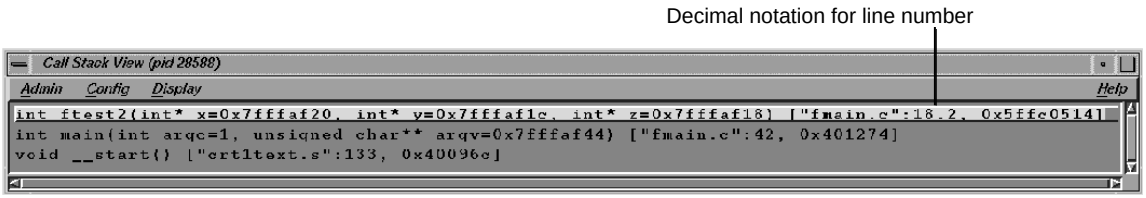

Figure A-66 Call Stack

Trap Manager

The **Trap Manager** recognizes redefined functions. It uses the decimal notation for line numbers, as shown in Figure A-67, page 313.

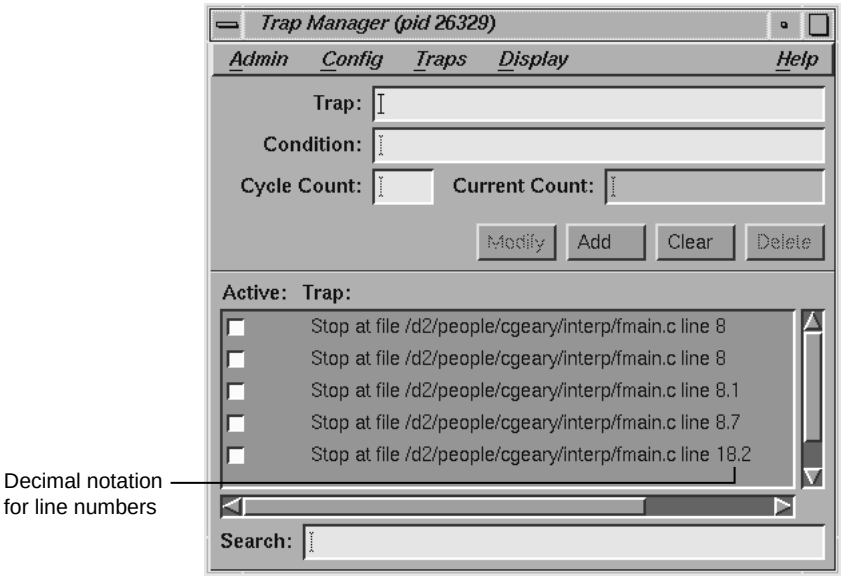

Figure A-67 Trap Manager Window with Redefined Function

Debugger Command Line

You can debug programs by entering `dbx`-style commands at the `cvd>` prompt found at the bottom of the Main View window (see Figure A-1, page 172). For more information, refer to the *dbx User's Guide*.

Syntax for dbx-style Commands

The syntax for the debugging commands is as follows:

`add_source` [*filename*`:`*line_number*]

(For C and C++ only, when compiled with `-o32`.) Prompts you to add source code lines (for example, `add_source "fmain.c":15.2`). *line_number* must be within the body of a function. Entering a period (.) specifies the end of your input. The source lines you provide are added after the specified line. This command returns an ID existing or new, depending on whether the function affected has already been changed or not. The resulting new

definition of the function is executed on its entry next time. See also `delete_source` and `replace_source`.

`alias` [*shortform command*]

Lists all aliases without arguments. With arguments, it assigns *command* to *shortform*.

`assign` *expression1*=*expression2*

Assigns *expression2* to *expression1*.

`attach` *pid*

Attaches to specified process ID (*pid*).

`call function_name`[*argument*`...`]

Executes the specified function with any arguments supplied.

`catch` [*signal_name* `|` `all`]

With no arguments, lists signals to be trapped. If a signal is specified, it's added to the list. If `all` is specified, it traps all signals.

`clearcalls`

Cancels pending function calls.

`cont in` *function_name*

Continues execution from the current line to the entry to the specified function.

`cont to` *line_number*

Continues execution from the current line until the specified line, if there is not an intervening breakpoint to stop execution.

`continue` [*all*]

Continues executing a program, or all processes, after a breakpoint. You can use both `c` and `cont` as aliases for `continue`.

`continue` [*signal*]

Sends specified signal and continues executing a program after a breakpoint.

`corefile` [*filename*]

With no arguments, reports whether data referencing commands reference a core file. If so, displays the current core file. With *filename* provided, specifies core file to be debugged.

`delete` [`all` | [*trap_number*] ,[*trap_number*], ...]

Deletes all traps. The `all` option deletes all breakpoints.

`delete_changes` [[*func_spec* | `-all`] | [`-file` *filename*]]

(For C and C++ only when compiled with `-o32`.) Deletes changes corresponding to the selected functions (for example, `delete_changes getNumbers -file fmain.c`). Once IDs are deleted, you will not be able to use the IDs again because the IDs associated with the selected functions are released. The default is `-all`. See also `save_changes`.

[`delete_source` *filename* :*line_number*] [,*line_number*]

(For C and C++ only, when compiled with `-o32`.) Deletes the given line(s) if *line_number* or *,line_number* (range) is within the body of a function. An example is:

```
delete_source "fmain.c":8.6,8.7
```

This command returns an ID existing or new, depending on whether the function affected has already been changed or not. The resulting new definition of the function is executed on its entry next time.

`delete` *trap_number* [,*trap_number*, ...]

Deletes the specified breakpoint numbered *trap_number*, as obtained from the list generated when the `status` command is issued.

`detach`

Detaches from the current process.

`disable all`

Deactivates all traps.

`disable_changes` [[*func_spec* `| -all`] `|` [`-file` *filename*]]

(For C and C++ only, when compiled with `-o32`.) Disables specified changes for selected functions (for example, `disable_changes getNumbers -file fmain.c`. Nothing happens if the selected function is already disabled. The compiled definition of the function is executed on its next entry. You can invoke this command when the process is stopped or on a running process when a function entry breakpoint is set.

`disable` *trap_number* [`,`*trap_number*`, ...`]

Deactivates the specified breakpoint numbered *trap_number*, as obtained from the list generated when the `status` command is issued.

`display` [*expression*`, ...`]

With *expression*, adds *expression* to the list of expressions displayed whenever the process stops. With no arguments, lists all expressions. See `undisplay` to delete an expression.

`down` [*number*]

Moves down the specified number of frames in the call stack. `down` moves away from the direction of the caller.

`dump`

Prints local variable values.

`enable all`

Reactivates all inactive traps.

`enable_changes` [*func_spec* `| -all`] `|` [`-file` *filename*]

(For C and C++ only when compiled with `-o32`.) Enables specified changes for selected functions (for example, `enable_changes getNumbers -file fmain.c`). Nothing happens if the selected function is already enabled. The latest accepted definition of the function is redefined on its next entry. You can invoke this command

when the process is stopped or on a running process when a function entry breakpoint is set.

`enable` *trap_number* [[`,`*trap_number*`, ...`]

Reactivates the specified breakpoint numbered *trap_number*, as obtained from the list generated when the `status` command is issued.

expression`/`[*count*] [*format*] or *expression*, [*count*] / [*format*]

Prints the contents of the memory address specified by *expression*, according to the specified format. *count* represents the number of formatted items. The following format options are available:

`d`	Prints a short word in decimal.
`D`	Prints a long word in decimal.
`o`	Prints a short word in octal.
`O`	Prints a long word in octal.
`x`	Prints a short word in hexadecimal.
`X`	Prints a long word in hexadecimal.
`b`	Prints a byte in octal.
`c`	Prints a byte as a character.
`s`	Prints a string of characters that ends in a null byte.
`f`	Prints a single-precision real number.
`g`	Prints a double-precision real number.

`file` [*filename*]

Displays the name of the current or specified file (*filename*). If a file is specified, it becomes the current file.

`func` [*func_name*]

Moves to the source code corresponding to the specified frame in the call stack or to the function in the executable if not on the stack.

`givenfile` [*filename*]

With no arguments, displays name of current object file. With *filename*, specifies object file to be debugged.

`goto` *linenumber*

Skips over lines going directly to the specified line number in the current routine. Unlike `dbx`(1), `cvd`(1) does not begin execution at the specified line.

`ignore`[*signal_name* `|` `all`]

With no arguments, lists those signals not to be trapped. If a signal is specified, this command removes it from the list of signals to be trapped. If `all` is specified, ignores all signals.

`kill`[*pid* `|` `all`]

Kills the specified process currently controlled by the Debugger or kills all processes.

`list`[*from-line#*[`:`*line_count*]`}` [`,`*to-line#*] `|` [*function_name*]]

Lists source lines beginning at *from-line#*. If no additional argument is specified, the default for *line_count* is `10`. If *line_count* is specified, a total of *line_count* lines are listed. If *function_name* is specified, the lines from the given function are listed.

`list_changes` [*func_spec* `|` `-all`] `|` [`-file` *filename*]

Lists one or more lines using the following syntax:

```
change_id isEnabled filename function_spec
```

For example:

```
4 enabled foo.c foo
8 disabled A.c++ A::bingo
```

The default is `list_changes -all`.

`next` [*int*]

Steps over the specified number of source lines. This command does not step into procedures. The default is one line.

`nexti` [*int*]

Steps over the specified number of machine instructions. This command does not step into procedures. The default is one instruction.

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

Prints the value of the specified expression(s). If the expression is a character pointer or array, both the string and address print. You can use `p` as an alias.

`printd` *expression* [`,`*expression*`, ...`]

Prints the value of the specified expression(s) in decimal format. You can use `pd` as an alias.

`printf` *string* [`,`*expression1* [`,` *expression2,* `]...`]

Prints the value(s) of the specified expression(s) in the format specified by the *string* string. The `printf` command supports all formats of the IRIX `printf` command except `%s`. For a list of formats, see the `printf`(1) man page.

`printo` *expression* [`,`*expression*`, ...`]

Prints the value of the specified expression(s) in octal format. You can use `po` as an alias.

`printregs`

Prints the contents of the registers.

`printx` *expression* [`,`*expression*`, ...`]

Prints the value of the specified expression(s) in hexadecimal format. You can use `px as an alias`.

`pwd`

Displays the current directory.

`quit`

Exits the debugging session.

`redefine` [*func_spec* `[-edit | -read]` *filename* [*line_number*`,`*line_number*]]

(For C and C++ only.) Specifies a new body for a function. The new definition is checked, and errors (if any) are printed. The new function body is redefined on the next function entry. Breakpoints (if set) on the old definition are put on the new definition based on their relative line number position from the beginning of the function definition. (Note that some breakpoints may not make it to the new definition.) You can invoke this command when the process is stopped or on a running process when a function entry breakpoint is set. There are three ways to provide a new definition:

- `-edit` pops up an editor of your choice containing the current definition of the function. The specification of the new definition is complete when you exit the editor. You may not leave the editor open.
- `-read` takes the contents of the file specified (within the line numbers if given) as the new function definition.
- No option allows you to type in replacement code from the next line. A period in the first column on a fresh line terminates the definition. For example:

```
redefine getNums
"/usr/fmain.c'':8.1> {
"/usr/fmain.c'':8.2> printf(''In getNums.\n'');
"/usr/fmain.c'':8.3> }
"/usr/fmain.c'':8.4> .
```

`replace_source` [*filename*`:`*line_number*`,`[*line_number*]]

(For C and C++ only when compiled with `-o32`.) Prompts you to type in replacement source if *line_number* or *,line_number* (range) is within the body of a function. The source lines you provide replace the specified line(s). An example is: `replace_source "fmain.c":12`. This command returns an existing or new id depending on whether the function affected has already been changed or not. The resulting new definition of the function is executed on its entry next time. See also `add_source` and `delete_source`.

`rerun`

Runs the program again using the same arguments.

`return`

Continues executing the current procedure and returns to the next sequential line in the calling function.

`run` [[`all`] | *argument_list*]

Runs the program(s). If an *argument_list* is specified, it is used as the arguments to be supplied to the program.

`runtime_check` *func_spec* [`-options` *key* [*key*`,...`]]

(For C and C++ only when compiled with `-o32`.) Enables all run-time checking options by default. If `-options` is specified, then run-time checking is restricted to the *keys*. The result of the runtime checks are printed the next time the specified function (*func_spec*) is entered. You can invoke this command on a stopped or a running process.

`save_changes` {*func_spec*`|` {`-file` *filename*}} [-[w | a]] *filename_to_save*

(For C and C++ only when compiled with `-o32`.) Saves (enabled or disabled) function redefinitions or an entire file to another file (*filename_to_save*). The following example shows how to save a function definition:

```
save_changes getNumbers getNumbersFunc
```

If you specify the `-file` option, then before saving to *filename_to_save*, all function changes are applied to the compiled source of the file (with the condition that the file has had only its functions redefined, and has not been edited since the last build). An example of saving an entire file is the following:

```
save_changes -file fmain.c fmain.c
```

The `-w` option replaces the *filename_to_save*. The `-a` option appends to the *filename_to_save*. An example of adding a function to a file is the following:

```
save_changes -file fmain.c -a fmain.c
```

See also `delete_changes`.

`setbuildenv` [*filename*] `compiler-flag-list`

(For C and C++ only, when compiled with `-o32`.) Overrides default build environment flags (compiler options). Without *filename*, the flags are passed along with `-c -g` flags to the compiler for any function in any file except those set separately with `setbuildenv`. An example is the following:

```
setbuildenv -DnameA -Idir
```

If *filename* is given, this command sets separate flags specifically for that file. For example, consider the following:

```
setbuildenv "fermat.c" -DnameB -Ianotherdir
```

See also `unsetbuildenv`.

`sh` [*shell_command*]

Calls a shell if no arguments; otherwise, executes the specified shell command.

`showbuildenv` [*filename*]

(For C and C++ only, when compiled with `-o32`.) Lists all the build environment flags set. `showbuildenv` with a *filename* lists any build environment specifications that have been set separately with `setbuildenv` "*filename*".

`show_changes` [*func_spec* `| -all |`[`-file` *filename*]]

(For C and C++ only, when compiled with `-o32`.) Prints the code of all enabled redefinitions of the specified function(s). The default is `show_changes -all`. See also `enable_changes` and `disable_changes`.

`show_diff` [*func_spec* `|` [`-file` *filename*]]

(For C and C++ only, when compiled with `-o32`.) Launches a `xdiff` comparing the compiled source and its latest redefinition for the specified function. If `-file` *filename* is specified, `xdiff` shows the difference between the compiled file and the file with all redefinitions applied to the compiled source of the file (with the condition that the file has had only its functions redefined, and has not been edited since the last build).

`showthread` [`full`] [`thread`] [*number* | `all`]

Shows brief status information about threads. If `full` is specified, prints full status information. You can request status information for a specific thread by number or you can request information for all threads. The `thread` qualifier does not affect command output.

`source` *filename*

Executes commands in the specified file.

`status`

Displays a list of currently set breakpoints and traces.

`step` [*int*]

Steps the specified number of source lines. This command steps into procedures. The default is one line.

`stepi` [*int*]

Steps the specified number of machine instructions. This command steps into procedures. The default is one instruction.

`stop` [`all` | `pgrp`] `in` [*filename*]

If set to `all`, stops all members of the specified process group whenever the trap is encountered. If set to `pgrp`, all members of the process group will apply the trap.

`stop at` [*file*] [*filename*] [*line*] [*line_number*] [`if` *expression*]

Traps at the specified line in the specified file. If the `if` option is used, the trap fires only if *expression* is true. If specified, the file name must be enclosed in double-quotation marks. As an example, to specify a stop at line `5` in `myfile.c`, the syntax is:

```
stop at "myfile.c":5
```

`stop exception` [`all` | `item`]}

Stops on all C++ exceptions or exceptions that throw the base type `item`. Do not include complex expressions using operators such as * and & in your type specification for an exception breakpoint.

`stop exception [all | [item] [, item]]`

Stops on all C++ exceptions or exceptions that either have no handler or are caught by an unexpected handler. If you specify `item`, stops on exceptions that throw the base type `item`. Do not include complex expressions using operators such as `*` and `&` in your type specification for an exception breakpoint.

`stop in [`*filename*`:] `*function_name*` [if `*expression*`]`

Traps at the entry to the specified function. If the `if` option is used, then the trap fires only if *expression* is true. If the *filename* is given, the function is assumed to be in that file's scope. If specified, the *filename* must be enclosed in double-quotation marks. As an example, to specify a stop in function `func1` in `myfile.f` only if *n* is `20`, the syntax is:

```
stop in "myfile.f":func1 if n .eq. 20
```

`syscall [catch | ignore] [call | return] [`*sys_call_name*` | all]`

The `catch` option adds a system call to the list of system calls to be trapped. The `ignore` option removes a system call from the system call trap list. The `call` option specifies the entry to the system call and `return` signifies the return from the call.

`[trace [`*variable*`] at `*filename* | *line_number*`] | `*function_name* \ `[if `*expression*`]`

Traces the specified variable. You can specify a file and/or test condition. You can also specify a line number or a function where the trace is to take place.

`unalias `*aliasname*

Cancels the alias specified as *aliasname*.

`undisplay [[`*displaynumber*`, ...]`

Stops display of expression with specified *displaynumber* when the process stops. Removes the expression from the display list.

`unsetbuildenv[`*filename*`]`

(For C and C++ only, when compiled with `-o32`.) Disregards the default build environment flags if specified earlier. For all functions

in files that don't have an overriding build environment, `unsetbuildenv` passes only the `-c` and `-g` flags.

If *filename* is given, this command disregards the build environment flags specified for the file earlier. Further redefinition of the functions in the file use the default build environment flags, if set. See also `setbuildenv`.

`up` [*number*]

Moves up the specified number of frames in the call stack. `up` moves in the direction of the caller.

`use` [*path*]

Uses the specified path to search for source files.

`watch` *identifier* [`write | read`]

Causes program to stop when the *identifier* is written or read, depending on whether `write` or `read` is specified. If neither is specified, the default is `write`.

`whatis` *identifier*

Displays all the qualifications of the specified variable.

`when at` [*filename*] [*line_number*] [*command*] [*command*] `...`

When your program reaches the specified *line_number*, the commands specified are executed before the program resumes execution.

`when in` [*filename*] *function_name* [*command*] [*command*] `...`

When your program enters the specified function, the commands specified are executed before the program resumes execution.

`which` [*identifier*]

Displays the qualification of the specified variable.

`where` [`thread` | *thread-id*]

Performs a stack trace showing the activation levels of a program or, optionally, of the specified thread. You can obtain *thread-ids* from the first column of output of the `showthread` command.

Blocking Kernel System Calls

The following are the kernel system calls (syscalls) that actually block continued pthreads. There are numerous library routines, such as `printf`, that can use one of these blocking system calls.

It would be impractical here to list all library routines which utilize a blocking syscall. Nevertheless, as a user you should know, for example, that if you call the `printf` library routine it eventually calls `writev()`, a blocking system call, and thus may block continued pthreads.

`accept`

accept a connection on a socket

`close`

close a file descriptor

`creat`

create a new file or rewrite an existing one

`dmi`

SGI specific. Used to implement the interface defined in X/Open document *Systems Management: Data Storage Managment (XDSM) API*.

`fcntl`

File and descriptor control. Provides for control over open descriptors.

`fsync`

synchronize a file's in-memory state with that on the physical medium

`getmsg` / `getpmsg`

get next message off a stream

`ioctl`

Control device. Performs a variety of control functions on devices and streams.

`lockf`

Record locking on files. Allows sections of a file to be locked.

`mq_open`

open/create a message queue

`msgsnd` / `msgrcv`

message send and message receive

`msync`

synchronize memory with physical storage

`nanosleep`

high resolution sleep

`open`

open for reading and writing

`pause`

suspend process until signal is received

`poll`

input/output multiplexing

`putmsg` / `putpmsg`

send a message on a stream

`read` / `readv` / `pread`

read from a file

`recv` / `recvfrom` / `recvmsg`

receive a message from a socket

`select`

synchronous I/O multiplexing

`semget` / `semctl` / `semop`

semaphore handling

`send` / `sendto` / `sendmsg`

send a message from a socket

`sginap`

times sleep and processor yield function

`write` / `writev` / `pwrite`

write on a file

Using the Build Manager

WorkShop lets you compile software without leaving the WorkShop environment. Thus, you can look for problems using the WorkShop analysis tools (Static Analyzer, Debugger, and Performance Analyzer), make changes to the source, suspend your testing, and run a compile. WorkShop provides two tools to help you compile:

- *Build View*—for compiling, viewing compile error lists, and accessing the code containing the errors in **Source View** (the WorkShop editor) or an editor of your choice. **Build View** helps you find files containing compile errors so that you can quickly fix them, recompile, and resume testing.
- *Build Analyzer*—for viewing build dependencies and recompilation requirements and accessing source files.

Build View uses the UNIX `make`(1) facility as its default build software. Although `cvmake` can be set up to run any program instead of make (for example, `gnumake`), `cvbuild` will only parse and display standard makefiles (in particular, it does not understand **gnu** make constructs).

Build View Window

You can access the **Build View** window from the WorkShop analysis tools, from the command line (by typing **`cvmake`**), or from the **Build Analyzer** (see next section).

To access **Build View** from WorkShop, select **Recompile** from the **Source** menu in the Main View window in the Debugger or from the **File** menu in **Source View** (for more information on the Main View and **Source View** windows, refer to Chapter 1, "WorkShop Debugger Overview", page 1). Selecting **Recompile** detaches the current executable from the WorkShop analysis tools and displays **Build View**. You can edit the **Directory** and **Target(s):** fields as needed and click **Build** to compile. If the source compiles successfully, the new executable is reattached when you reenter the WorkShop analysis tools.

The **Build View** window has three major areas:

- "Build Process Control Area", page 330
- "Transcript Area", page 331
- "Error List Area", page 331

Build Process Control Area

The build process control area lets you run or stop the build and view the status. See Figure B-1.

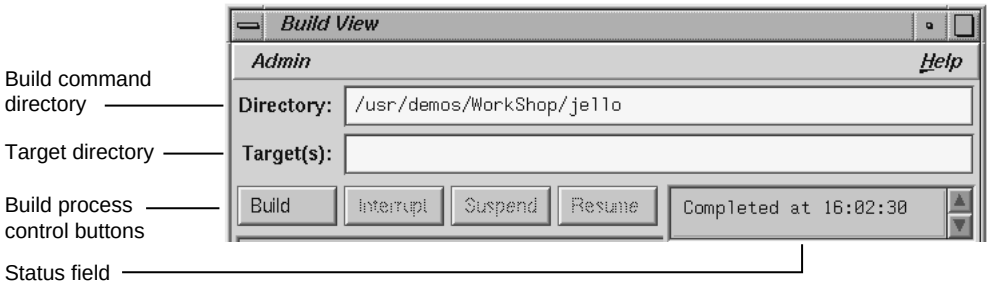

Figure B-1 Build Process Control Area in **Build View** Window

The directory in which the build will run displays in the **Directory:** field at the top of the area. The current directory displays by default. You can specify the build using `make`, `smake`, `pmake`, `clearmake`, or any other builder and any flags or options that the builder understands (see "Build View Preferences", page 332, and "Build Options", page 333). The target to be built is specified in the **Target(s):** field.

The build process control buttons let you control the build process. The following buttons are available:

Build	Runs (or reruns) a build. If you have modified any files you will be prompted to save the new versions prior to the compile.
Interrupt	Stops a build.
Suspend	Stops a build temporarily.
Resume	Restarts a suspended build.

The status field is to the right of the build process control buttons. It indicates the progress of the build.

Transcript Area

The transcript area displays the verbatim output from the build. The vertical scroll bar lets you go through the list; the horizontal scroll bar lets you see long messages obscured from view. A sash between the compile transcript area and the error list area lets you adjust the lengths of the lists displayed. See Figure B-2.

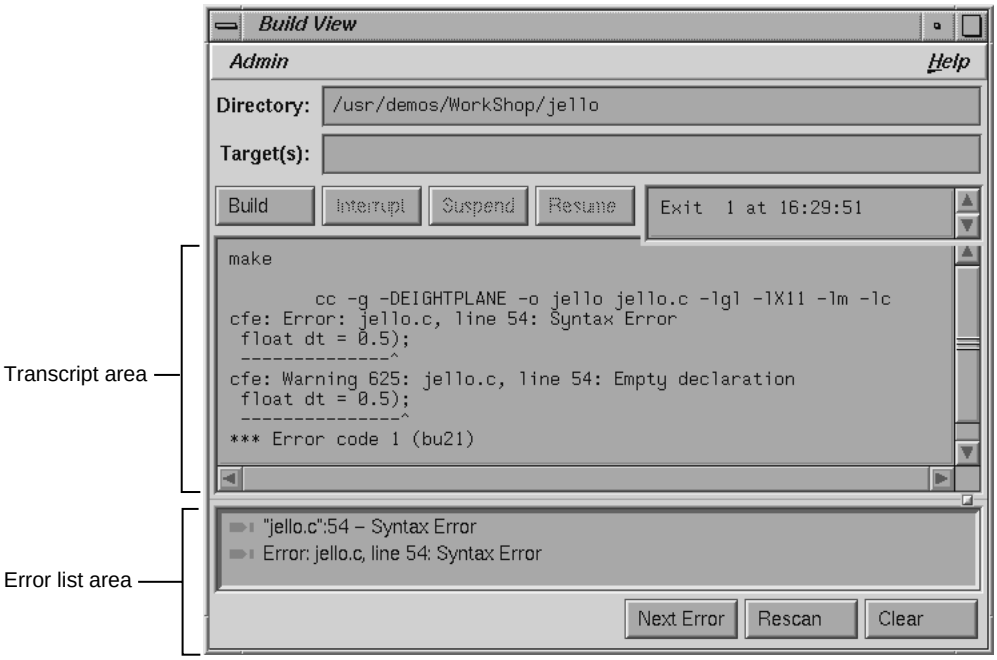

Figure B-2 Build View Window with Typical Data

Error List Area

The error list area consists of the error list display and three control buttons. The following buttons are available:

Next Error	Brings up the default editor scrolled to the next error location. This button is below the error list display.
Rescan	Refreshes the error list display.

Clear Clears the error list display area.

The error list area displays compile errors (see Figure B-2, page 331). The errors are annotated according to their severity level (fatal has a solid icon and the warning icon is hollow). Double-clicking the text portion of an error brings up the default editor scrolled to the error location and displays a check mark to help you keep track of where you are in the error list. Check marks also display when you click the **Next Error** button.

Build View Admin Menu

The **Admin** menu in **Build View** has two selections in addition to the standard WorkShop entries:

- "Build View Preferences", page 332
- "Build Options", page 333

For information on **Launch Tool**, **Project**, and **Exit** menu selections, refer to "Admin Menu", page 179.

Build View Preferences

The **Preferences** selection brings up the dialog box shown in Figure B-3, page 333. The options are:

Maker Program field

Lets you enter the program you use to build your executable.

Macro Settings field

Lets you enter build macros, such as

```
CFLAGS=-g.
```

Makefile field

Lets you enter the name of a makefile if you do not wish to use the default.

Discard Duplicate Errors button

Eliminates subsequent duplicates of errors in the error list area.

Show Warnings button

Toggles the option to display warnings in the list.

Figure B-3 Build View Preferences Dialog

Build Options

The **Build Options Dialog** lets you add the options shown in Figure B-4, page 334, to your `make` command.

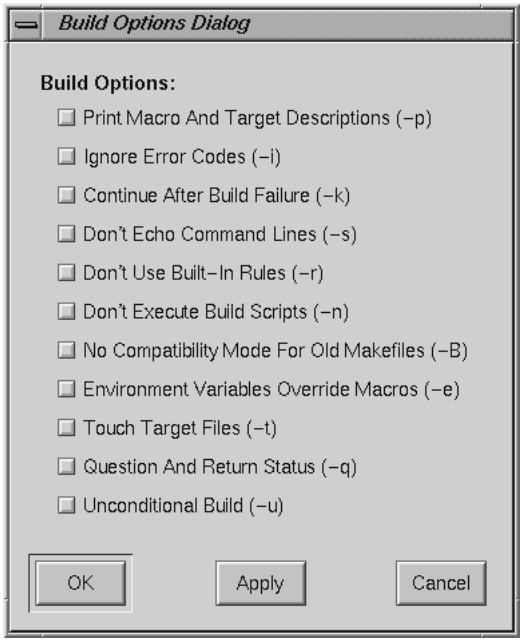

Figure B-4 Build Options Dialog

Using Build View

The steps in running a compile using **Build View** are as follows:

1. Bring up the **Build View** window.
2. Edit the **Target(s):** and **Directory:** fields as required.
3. Specify your preference regarding duplicate errors and warnings using the **Admin** menu (optional).
4. Click **Build** to start the build. All compile information displays in the transcript area. Errors are grouped in a list below.
5. Click **Interrupt** to terminate or **Suspend** for a temporary stop, if you want to stop the build. The **Resume** button restarts a suspended build.
6. Double-click an error to bring up your preferred editor with the appropriate source code. A check mark indicates that an error has been accessed.

Note: The default editor is determined by the `editorCommand` resource in the `app-defaults` file. The value of this resource defaults to `wsh -c vi +%d`, which means run `vi` in a `wsh` window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.

7. Click **Build** to restart the build.

Build Analyzer Window

The **Build Analyzer** window displays a graph indicating the source files and derived files in the build, and their dependency relationships and current status. Source files refers to input files, such as code modules, documentation, data files, and resources. Derived files refers to output files, such as compiled code. Your request builds in **Build Analyzer** by either:

- Double-clicking a derived module
- Making a selection from the **Build** menu

You access **Build Analyzer** from WorkShop by selecting **Launch Tool** from the **Admin** menu in Main View. Outside of WorkShop, you can access **Build Analyzer** by typing **`cvbuild`** at the command line. A typical **Build Analyzer** window appears in Figure B-5, page 336, with the menus displayed.

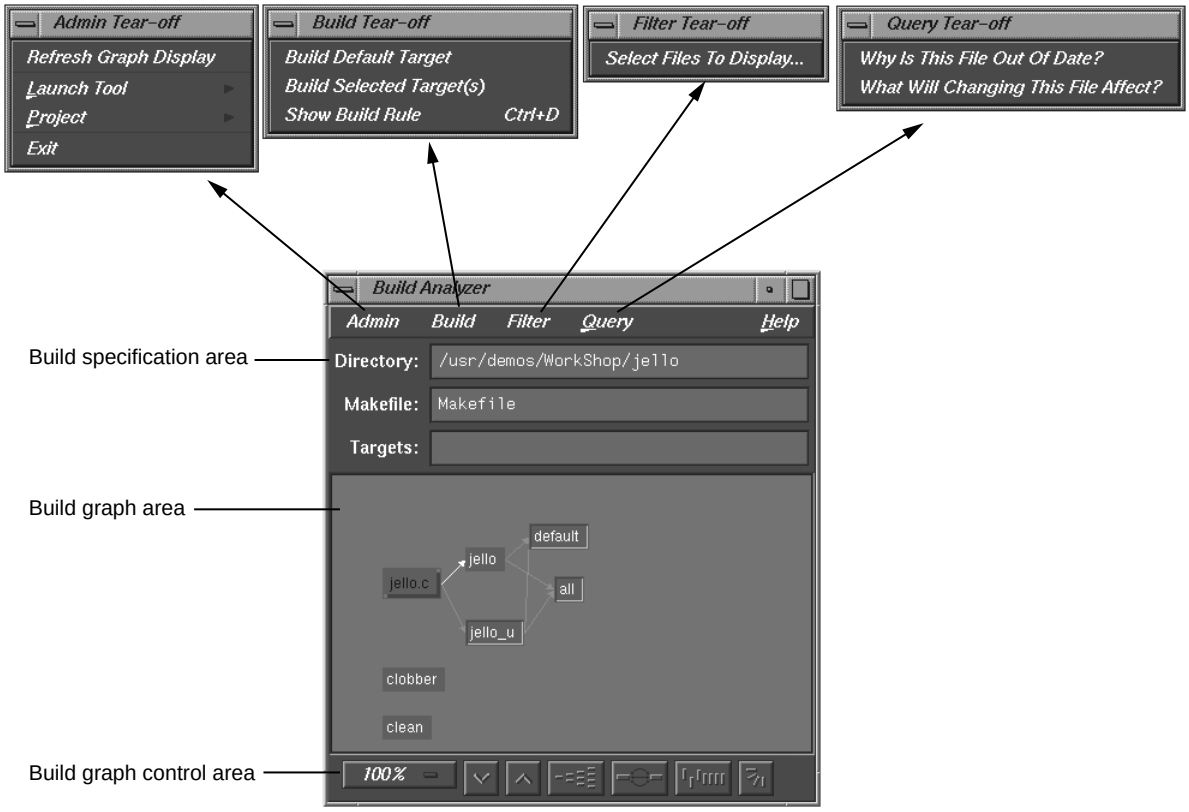

Figure B-5 Build Analyzer Window

Build Specification Area

The three fields in the build specification area identify the working directory, makefile script, and target file(s) for compilation. You can edit the **Directory**, **Makefile**, and **Targets** fields directly. The **Targets** field also lets you specify a search string for locating a file in the build graph.

Build Graph Area

The build graph area displays the specified source and derived files and their dependency relationships. Files are depicted as rectangles; dependency relationships are shown as arrows, with the supplying file at the base of the arrow and the dependent file at the head. The colors used to depict the files depends on your color scheme. **Build Analyzer** differentiates the two types of files by depicting one with light characters on a dark background and the other with dark text on a light background. If you double-click a source file icon, an editor is brought up for that file. Double-clicking a derived file starts a build and displays Build View.

In addition to dependency relationships, Build Analyzer indicates the status of the files and relationships as follows:

- Source file availability status: `normal` or `checked out`
  - Normal means that the source file is read-only and needs to be made writable to be edited. Normal files appear as light rectangles with black text.
  - `Checked out` means that you have a writable version of this file available and can thus edit it. A checked out file appears in a different color (from normal files) with a shadow.
- Derived file compile status: `current` or `obsolete`
  - When applied to a derived file, the term current means that none of the files on which the derived file depends have been edited since the derived file was created. Current derived files appear as dark rectangles with white text.
  - `Obsolete` means that one or more of the source files have been modified since the derived file was created. Obsolete files appear in the same color as current derived files but with a colored outline.
- Dependency relationship: `current` or `obsolete`
  - `Current` means that the derived file is up to date with the source files. Note that a relationship can be current even if both files are obsolete. This happens when a file on which both files are dependent has been modified. Current arcs are black.
  - `Obsolete` means that the source file has changed and the derived file has not been updated accordingly. Obsolete arcs appear as colored arrows.

Some typical build graph icons are shown in Figure B-6, page 338.

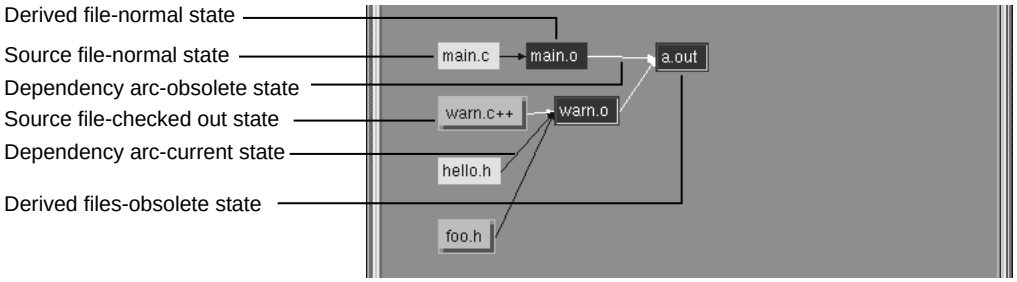

Figure B-6 Build Graph Icons

The `main.c` and `hello.h` source files are in their normal state. The source files `warn.c++` and `foo.h` are checked out and thus appear highlighted and with dropped shadows. The derived file `main.o` is current, since it has not changed since the last compile. The black dependency arcs indicate that the source and derived files at either end are current with each other. When an arc is highlighted, it indicates that the source has changed since the last compile. The derived files `warn.o` and `a.out` are obsolete because `warn.c++` has changed.

Build Graph Control Area

The build graph control area contains a row of graph control buttons similar to the ones in the WorkShop Static Analyzer and the **Call Graph View** in the Performance Analyzer. The **Overview** button is particularly useful in the **Build Analyzer** because it helps you quickly find obsolete files where a lot of dependencies are involved.

The build graph control area is shown in Figure B-7, page 339.

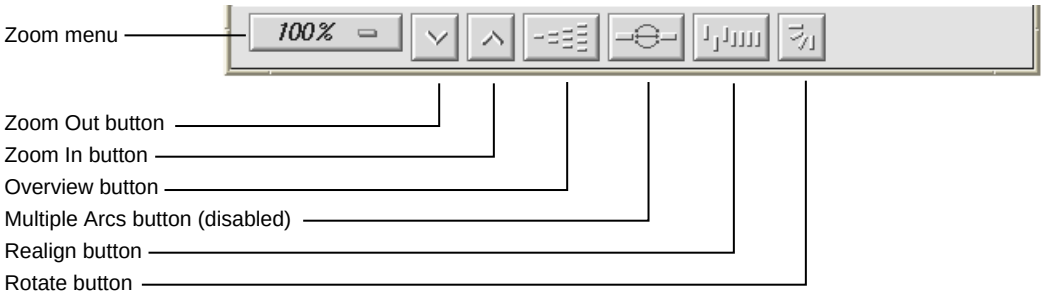

Figure B-7 Build Graph Control Area

Build Analyzer Overview Window

Since build graphs can get quite complicated, an overview mode (similar to those in Static Analyzer and Profiling View) is supplied that lets you view the entire graph at a reduced scale. To display the overview window, you click the **overview** icon (see Figure B-7, page 339).

Figure B-8, page 340, shows a typical **Build Analyzer Overview** window with the resulting graph. The window has a movable viewport that lets you select the portion of the build graph displayed in **Build Analyzer**. Source files that have changed and derived files needing recompilation are highlighted for easy detection. In this particular color scheme, the **Build Analyzer Overview** window displays normal source files in turquoise, checked out source files in pink, current derived files in dark blue, and obsolete derived files in yellow. Arcs appear only in black in this window.

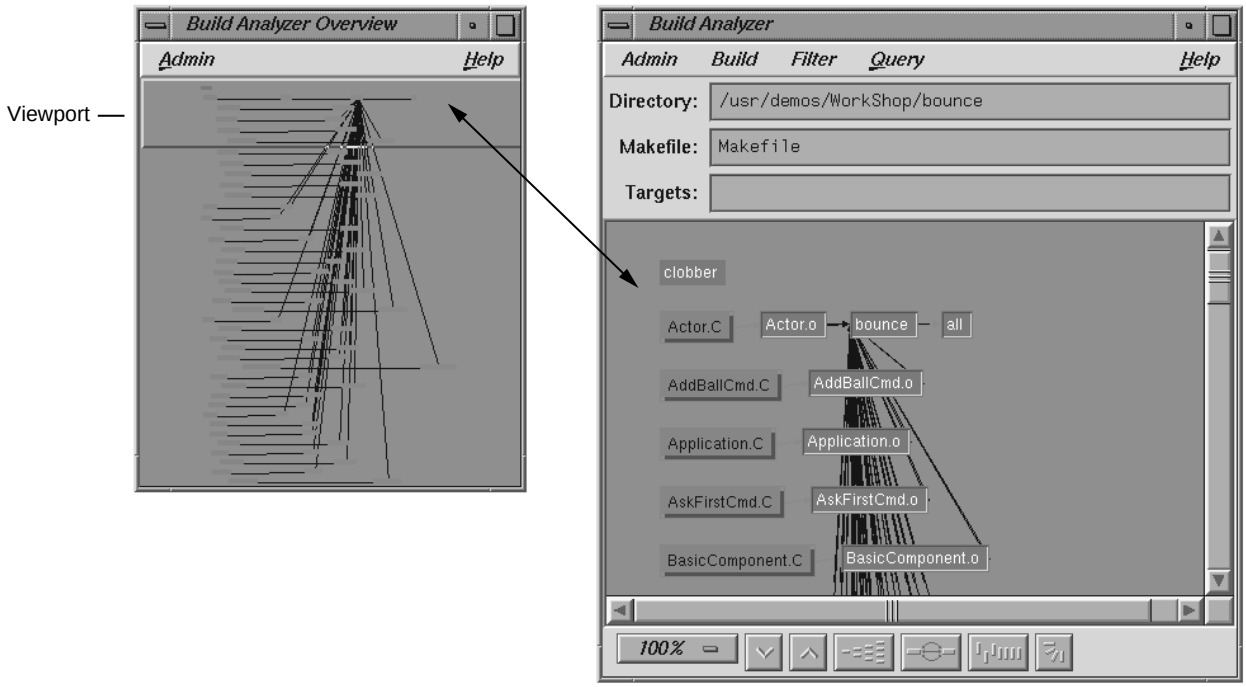

Figure B-8 Build Analyzer Overview Window with Build Analyzer Graph

Build Analyzer Menus

The **Build Analyzer** window contains the following menus:

- Admin
- Build
- Filter
- Query

Admin Menu

The **Admin** menu provides one selection **Refresh Graph Display** in addition to the standard WorkShop selections.

Refresh Graph Display

Refreshes the window.

Launch Tool

Lets you execute the WorkShop tools. For more information, see the "Admin Menu", page 179.

Project

Lets you control the WorkShop tools operating on the same executable as a group. For more information, see "Admin Menu", page 179.

Build Menu

The selections in the **Build** menu let you perform builds as follows:

Build Default Target

Performs a make with no arguments.

Build Selected Target(s)

Performs the build(s) as entered in the **Target(s):** field.

Show Build Rule

Displays a dialog box showing the makefile line for the selected node.

Filter Menu

The **Filter** menu has only one selection:

Select files to show in graph

Opens the **File Filter** dialog box that lets you enter a regular expression to filter files displayed in the build graph.

The upper list area lets you specify files to be excluded from the build graph. The lower list is for specifying files to appear in the graph.

Query Menu

The **Query** menu lets you request information about the build graph. The following selections are available:

Why Is This File Out Of Date?

Identifies the source files requiring this file to be recompiled. This query only applies to derived files.

What Will Changing This File Affect?

Shows all derived files dependent on this source file.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X