ProDev™ WorkShop: Debugger User's Guide

007–2579–009

New Features in this Guide

User level preferences to control the setting of internal breakpoints have been added. See Chapter 10, "Multiple Process Debugging", page 117 for details.

Information regarding data diving on arrays has been incorporated into Chapter 1, "WorkShop Debugger Overview", page 1.

Some of the screen representations in this guide may not be current.

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.
008	September 2002 Supports the ProDev WorkShop 2.9.2 release.
009	June 2003 Supports the ProDev WorkShop 2.9.3 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 (1.4.2 and older versions), 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 43, presents a short Debugger tutorial based on demonstration programs provided with your WorkShop tools.
- Chapter 5, "Setting Traps (Breakpoints)", page 61, describes how to set various types of traps.
- Chapter 6, "Controlling Program Execution", page 77, describes methods for controlling process execution.
- Chapter 7, "Viewing Program Data", page 81, explains how to examine Debugger data.
- Chapter 8, "Debugging with Fix+Continue", page 91, presents a short tutorial using Fix and Continue.
- Chapter 9, "Detecting Heap Corruption", page 107, describes heap corruption problems and how to detect them.
- Chapter 10, "Multiple Process Debugging", page 117, describes debugging multiprocess programs.
- Chapter 11, "X/Motif Analyzer", page 169, presents a short tutorial using the X/Motif Analyzer.

- Chapter 12, "Customizing the Debugger", page 185, gives you tips on how you can customize the Debugger to the requirements of your working environment.
- Appendix A, "Using the Build Manager", page 191, 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://docs.sgi.com`

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.
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 publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI 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 Web page:

  `http://docs.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 Parkway, M/S 535
  Mountain View, California 94043–1351

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

SGI values 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 (1.4.2 and older versions), 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", page 1
- "Debugging with Fix+Continue", page 8
- "Debugging with the X/Motif Analyzer", page 8
- "Customizing the Debugger", page 8

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:

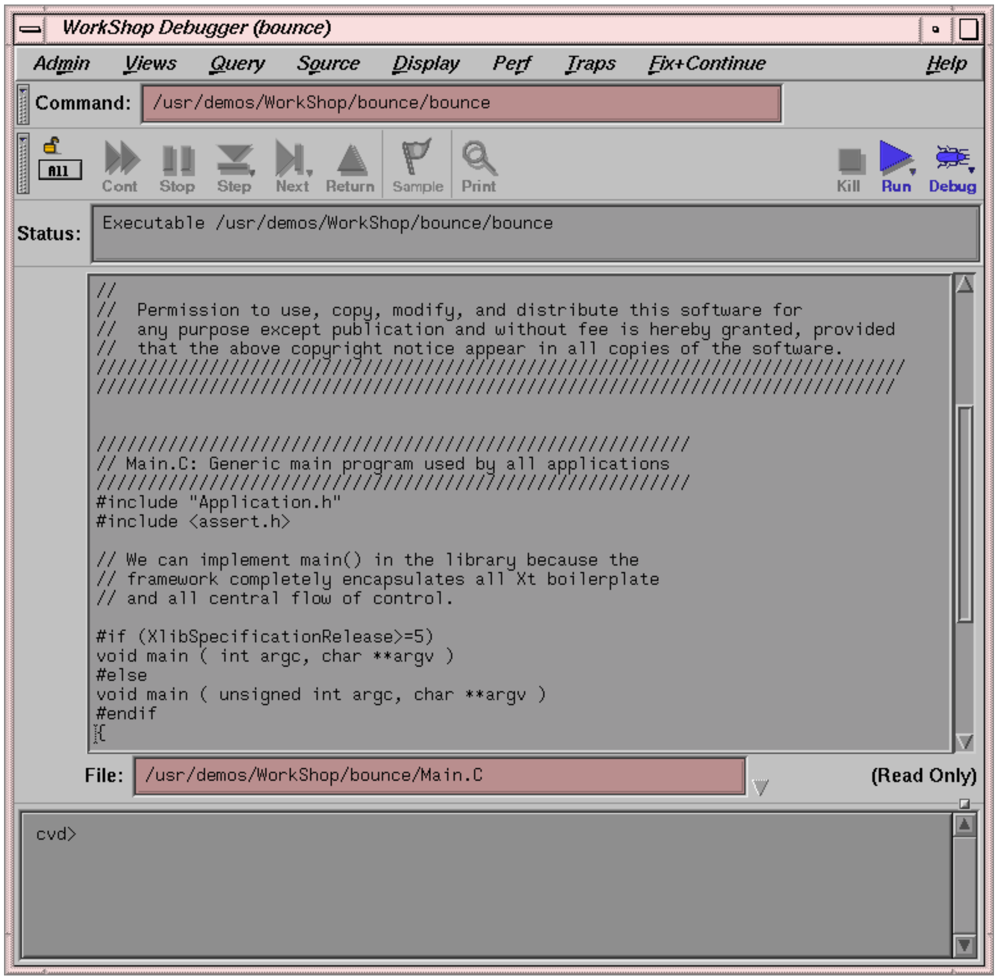
- "The Debugger Main View Window", page 1
- "About Traps", page 3
- "Viewing Program Data", page 4
- "Integrating the Debugger with Other WorkShop Tools", page 5
- "Using the Mouse for Data Diving", page 7

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 the *ProDev WorkShop: Debugger Reference Manual* 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 the *ProDev WorkShop: Debugger Reference Manual*.



Command: /usr/demos/WorkShop/bounce/bounce

Status: Executable /usr/demos/WorkShop/bounce/bounce

```
//
//  Permission to use, copy, modify, and distribute this software for
//  any purpose except publication and without fee is hereby granted, provided
//  that the above copyright notice appear in all copies of the software.
/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////
// Main.C: Generic main program used by all applications
////////////////////////////////////////////////////////////
#include "Application.h"
#include <assert.h>

// We can implement main() in the library because the
// framework completely encapsulates all Xt boilerplate
// and all central flow of control.

#if (XlibSpecificationRelease>=5)
void main ( int argc, char **argv )
#else
void main ( unsigned int argc, char **argv )
#endif
{
```

File: /usr/demos/WorkShop/bounce/Main.C

```
cvd>
```

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 (Breakpoints)", page 61.

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:

- **Array Browser**: this option allows you to inspect the values of an array variable.
- **Call Stack**: this option allows you to inspect the call stack at the breakpoints.
- **Data Explorer**: this option allows you to inspect data structure.
- **Data View**: this option allows you to interactively inspect separate data structures in a view. Pointers can be followed by *diving* (right mouse click) on the field to dereference it.
- **Disassembly View**: this option allows you to inspect the disassembled code.

- **Expression View**: this option allows you to inspect the value of specified expressions.
- **Memory View**: this option allows you to inspect the values in specified memory locations.
- **Multiprocess View**: this option allows you to inspect the values of multiple and / or pthreaded processes.
- **Register View**: this option allows you to inspect registers.
- **Variable Browser**: this option allows you to inspect the values, types, or addresses of variables.

You can also view data by using the data diving techniques described in "Using the Mouse for Data Diving", page 7.

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 191.

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 197.

For general information on the Build Manager tools, see Appendix A, "Using the Build Manager", page 191.

Using the Mouse for Data Diving

Dynamic menus are available in all of the Debugger views and windows. To access the dynamic menus, position the mouse over the item you are interested in and click the right mouse button. A menu of actions that can be done with that object is displayed.

Data diving is available in several debugger views and screens. Click the right mouse button over any item and a default action occurs (when appropriate).

Depending on the cursor location, the result of a mouse action will vary:

- In the **Source View**, a click of the mouse button over a variable displays the variable in the Data View window. Holding a mouse button down brings up a menu of available actions.
- Holding a mouse button down over a blank area in a view brings up a menu that allows you to **Jump To** the selected line or **Continue To** the selected line.
- When in the icon canvas area (the part of the screen to the left of the source code listing), holding a mouse button down provides a list of actions that can be performed against that line of code (for example, set and delete traps, enable or disable traps, or access the trap manager). Clicking the mouse in the icon canvas area sets or deletes a breakpoint.
- A right mouse button click on a user function in the **Source View** displays the source for that function.

Data Diving on Arrays

You can use the data diving feature described previously to obtain information about arrays in your program. When you dive on an array, the **Data View** window appears.

From this window you can view statistics for the array (dimensions, total number of elements, maximum and minimum value), show the entire array, and 'slice' the array.

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 91, 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 169 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 185, for more tips on customizing the Debugger to your specific needs.

Chapter 2

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 run, 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 automatically appears 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 pops 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 the *ProDev WorkShop: Debugger Reference Manual* 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.

Also see the *ProDev WorkShop: Debugger Reference Manual* 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 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 enables 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 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 enables 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 137.)
- **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 is automatically 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 Single** applies control action to all processes or threads if the button is set to **All**. If set to **Single**, 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 (Breakpoints)

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 is highlighted in red. To remove the breakpoint, click on the red highlight in the left canvas and the breakpoint disappears. Clicking on the **Run** button causes 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.

Using the Mouse for Data Diving

Data diving is available in several debugger views and screens. Click the right mouse button over any item and a default action occurs (when appropriate). Depending on the cursor location, the result of a mouse action will vary. For example:

- In the **Source View** window, a click of the right mouse button on a variable displays that variable in a **Data View**. Holding the mouse button down brings up a menu of available actions.
- A right mouse button action on a user function in the **Source View** displays the source for that function.
- Holding a mouse button down over a blank area in a view brings up a menu that allows you to **Jump To** the selected line or **Continue To** the selected line.
- In the icon canvas area (to the left of the source code listing), a right mouse button action sets and unsets a breakpoint. Holding down the mouse button provides a list of actions that can be performed against the line of code (for example, set or delete a trap, enable or disable a trap).

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* are 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) is now 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 21).

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` are 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 the *ProDev WorkShop: Debugger Reference Manual*.

Viewing Variables Using `Click To Evaluate`

Perform the following to view variables with `Click To Evaluate` (available in the **Display** Menu):

1. Right-click in the window that contains your source code to bring up the menu.
2. Select **Click To Evaluate** from the menu. You can now click on any variable and its value appears. For example:
   - If you click on the `x` in `x(i)`, the address of `x` appears.
   - 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.

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 are 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**.

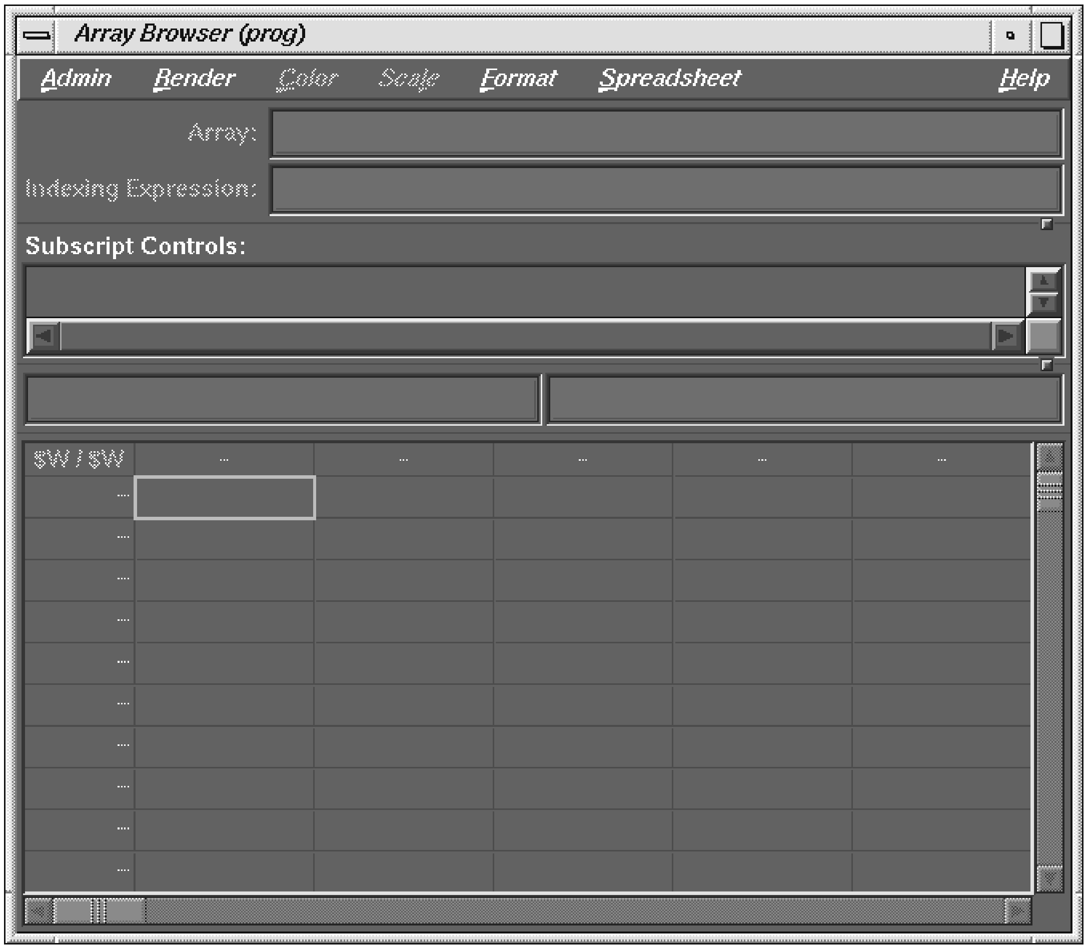


Figure 2-1 Array Browser Window

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

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

The current values of, at most, two dimensions of the array display in the lower pane of the **Array Browser** window. The values of the array are 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 > Data Explorer** from the Main View window menu bar to call up the **Data Explorer** 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 brings up a

new window showing what is being pointed at and an arrow appears showing this relationship. This can aid in debugging linked lists, for example.

There is another way to do this without using the Data Explorer. If the user performs a mouse click on the variable name (“dives” on the variable name) in the source view, a Data View is dislayed, showing the structure. Click the right mouse button to access the menu and to have the variable displayed or added to the **Data Explorer**.

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**.
2. Enter the character string for which you would like to search in the search utility 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 appears 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.

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

You can also "dive" on the Call Stack. A dive on the function name performs the same function as a double-click. Diving on the arguments brings up a Data View window with that argument displayed. Additionally, a Dive Menu is displayed if a right-mouse button action is performed.

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 these ways:

- Use the "data diving" method described in "Using the Mouse for Data Diving", page 7, to dive on the function name in the Source View and then click the mouse to select a breakpoint. You can also use a right-mouse button action over the function name to bring up a menu and then select **Stop In Function** from the menu.

- 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 stops 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...
```

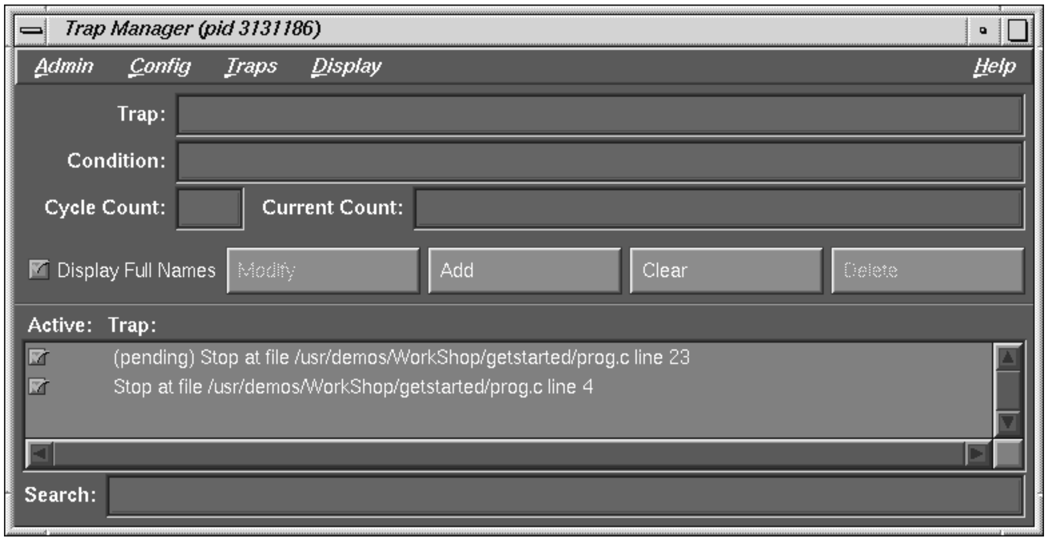
Here, the value of *n* is a positive integer and *name* is the name of the function or subroutine where the stop has been set.

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.



```
(pending) Stop at file /usr/demos/WorkShop/getstarted/prog.c line 23
Stop at file /usr/demos/WorkShop/getstarted/prog.c line 4
```

Figure 2-2 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 does 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 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:

- "Step 1: Use `lint`", page 28
- "Step 2: Check for Out-of-Bounds Array Accesses", page 28
- "Step 3: Check for Uninitialized Variables Being Used in Calculations", page 29
- "Step 4: Find Divisions by Zero and Overflows", page 30
- "Step 5: Perform Core File Analysis", page 31
- "Step 6: Troubleshoot Incorrect Answers", page 32

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 helps 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 stops 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 executes under `cvd`. It also disables optimization. Sometimes, disabling optimization causes 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 causes the source code for this function or subroutine to display in the Main View window, and the line where `cvd` stops is 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 causes the source code for this function or subroutine to display in the Main View window and the line where the Debugger stopped is 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` is not detected when `y` is uninitialized since no floating point calculations are being done. However, the following are 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 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 brings 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 is 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 brings up the source code for this routine, and the last line executed is 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, can 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", page 35
- "Path Remapping", page 38

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
various messages appear
% 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
  ```

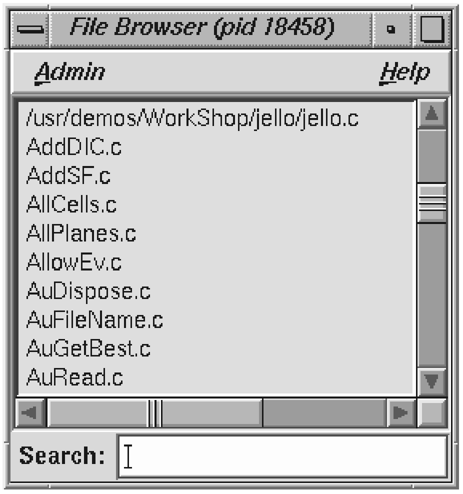
- Enter the executable directly. Enter the name (or full pathname, if necessary) of the executable in the **Command** field.

  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.



```
/usr/demos/WorkShop/jello/jello.c
AddDIC.c
AddSF.c
AllCells.c
AllPlanes.c
AllowEv.c
AuDispose.c
AuFileName.c
AuGetBest.c
AuRead.c
```

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.

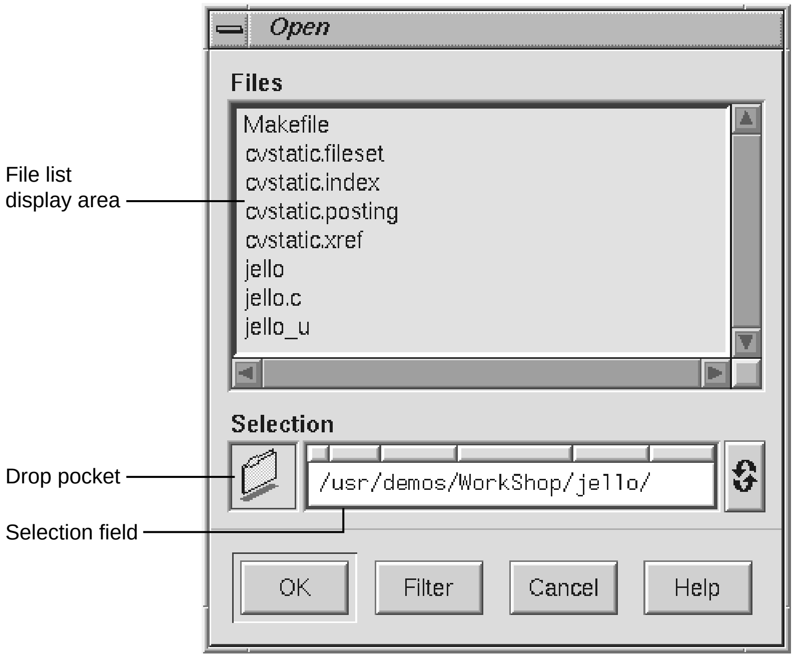


Figure 3-2 Open Dialog Box

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 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:

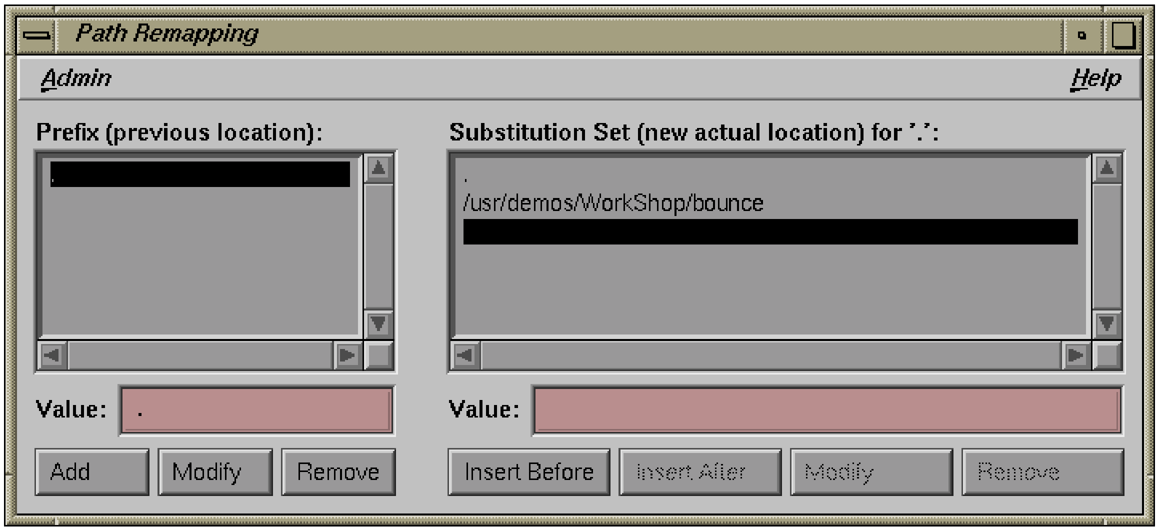

Path Remapping
Admin
Help
Prefix (previous location):
Substitution Set (new actual location) for '.':
.
/usr/demos/WorkShop/bounce
Value:
.
Value:
Add
Modify
Remove
Insert Before
Insert After
Modify
Remove

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 displays 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* appears 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 44, 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 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).

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 4-1, page 46). 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.

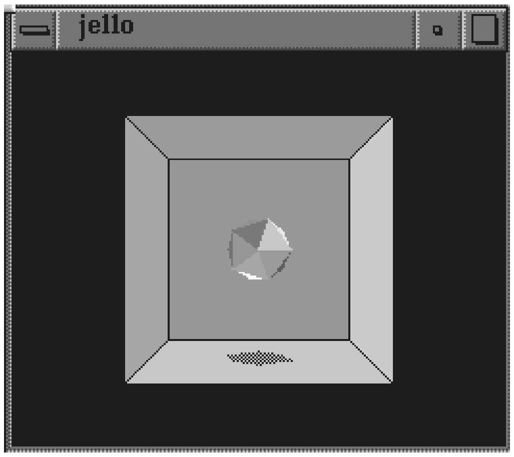


Figure 4-1 The `jello` Window

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.

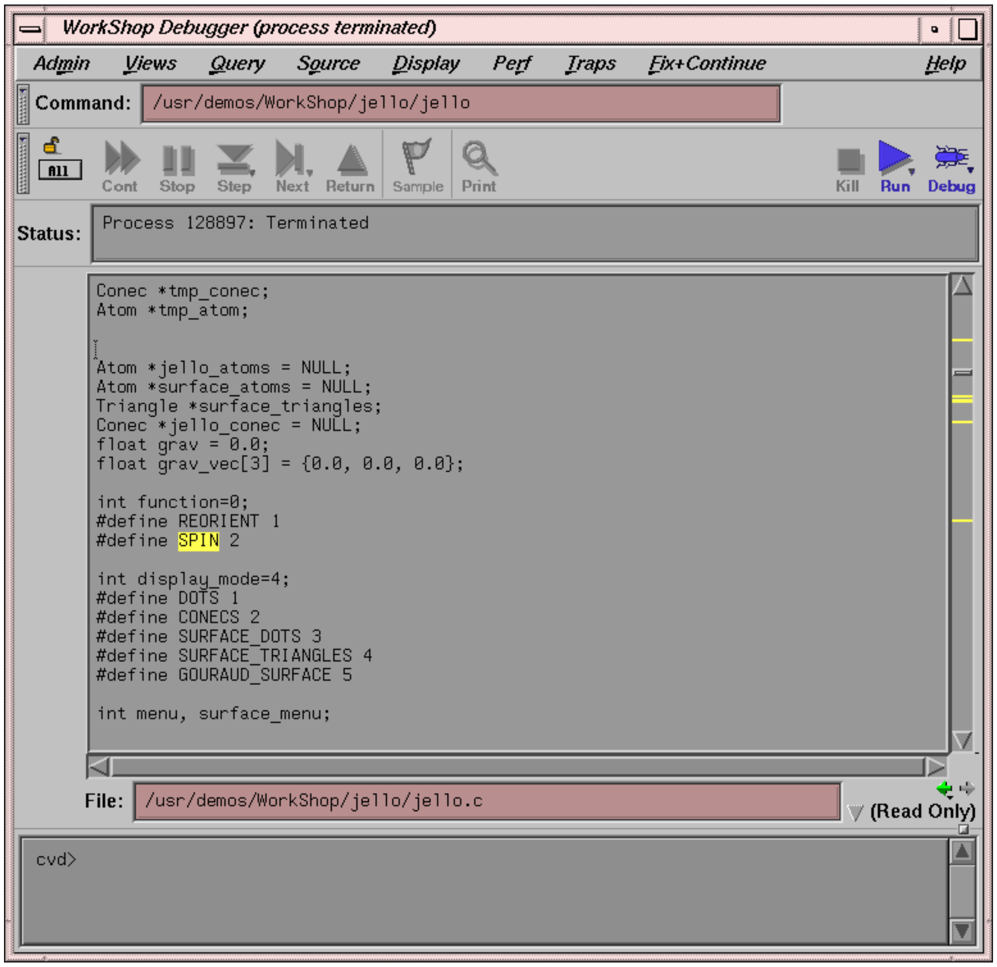
3. Click the **Apply** button.

   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 the search target indicators on the right side of the screen in Figure 4-2, 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 information about the **Source** menu in the *ProDev WorkShop: Debugger Reference Manual*.

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.



Command: /usr/demos/WorkShop/jello/jello

Status: Process 128897: Terminated

```
Conec *tmp_conec;
Atom *tmp_atom;


Atom *jello_atoms = NULL;
Atom *surface_atoms = NULL;
Triangle *surface_triangles;
Conec *jello_conec = NULL;
float grav = 0.0;
float grav_vec[3] = {0.0, 0.0, 0.0};

int function=0;
#define REORIENT 1
#define SPIN 2

int display_mode=4;
#define DOTS 1
#define CONECS 2
#define SURFACE_DOTS 3
#define SURFACE_TRIANGLES 4
#define GOURAUD_SURFACE 5

int menu, surface_menu;
```

File: /usr/demos/WorkShop/jello/jello.c

```
cvd>
```

Figure 4-2 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` determines 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 A, "Using the Build Manager", page 191.

   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 (Breakpoints)", page 61.

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

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

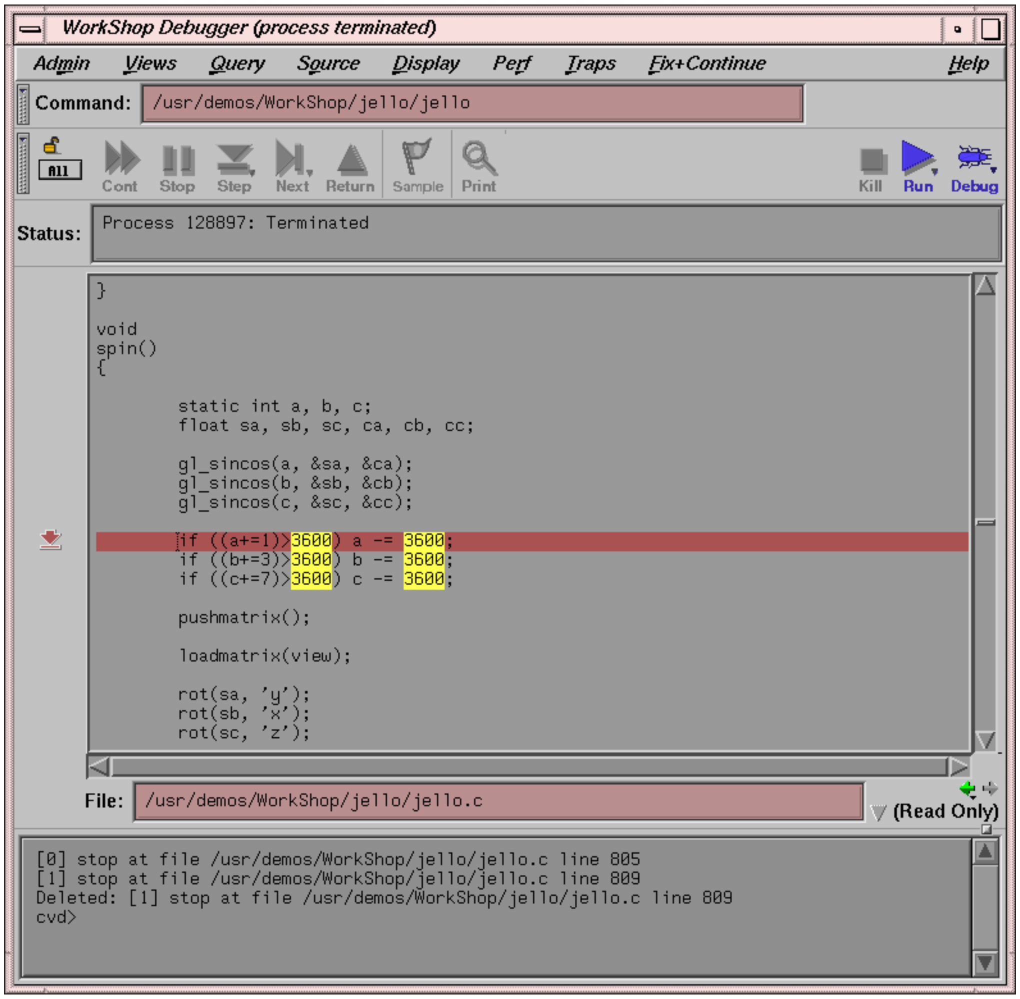
   The demo window displays.

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 4-3, 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.



```
/usr/demos/WorkShop/jello/jello
```

```
Process 128897: Terminated
```

```
}

void
spin()
{

        static int a, b, c;
        float sa, sb, sc, ca, cb, cc;

        gl_sincos(a, &sa, &ca);
        gl_sincos(b, &sb, &cb);
        gl_sincos(c, &sc, &cc);

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

        pushmatrix();

        loadmatrix(view);

        rot(sa, 'y');
        rot(sb, 'x');
        rot(sc, 'z');
```

```
/usr/demos/WorkShop/jello/jello.c
```

```
[0] stop at file /usr/demos/WorkShop/jello/jello.c line 805
[1] stop at file /usr/demos/WorkShop/jello/jello.c line 809
Deleted: [1] stop at file /usr/demos/WorkShop/jello/jello.c line 809
cvd>
```

Figure 4-3 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.

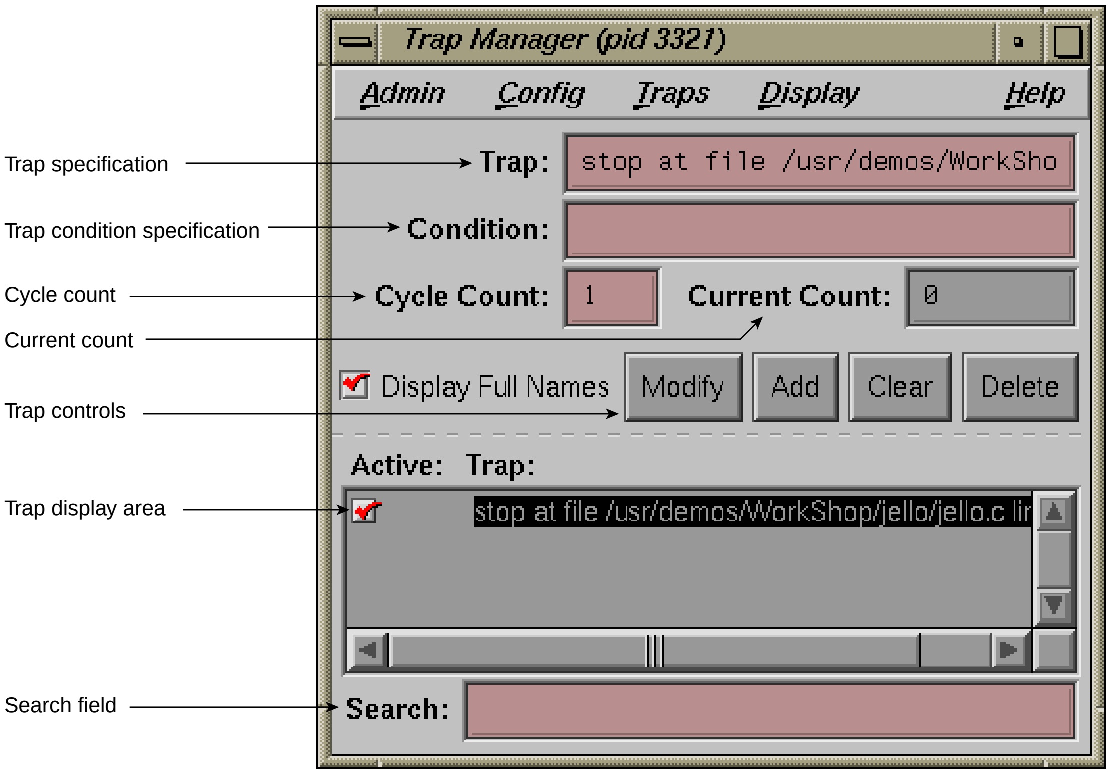
5. Select **Views > Trap Manager** from the Main View window menu bar.

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



```
Trap: stop at file /usr/demos/WorkSho
Cycle Count: 1
Current Count: 0
stop at file /usr/demos/WorkShop/jello/jello.c lir
```

Figure 4-4 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 is 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. Note that you can also examine data using the data diving techniques described in "Using the Mouse for Data Diving", page 7.

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

   The **Call Stack** window appears. 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 81.

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.

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).

   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.

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. 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.

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 84.

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 lets you apply the language semantics to the expression.

   The **Format** menu (shown on the right side of the **Expression View** window) lets you view the value, type, address, or size of the result. You can further specify the display format for the value and address.

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 > Data Explorer** from the Main View window to open the **Data Explorer**.

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

    The **Data Explorer** 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 **Data Explorer** de-references 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 now appears.

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 **Data Explorer** window.

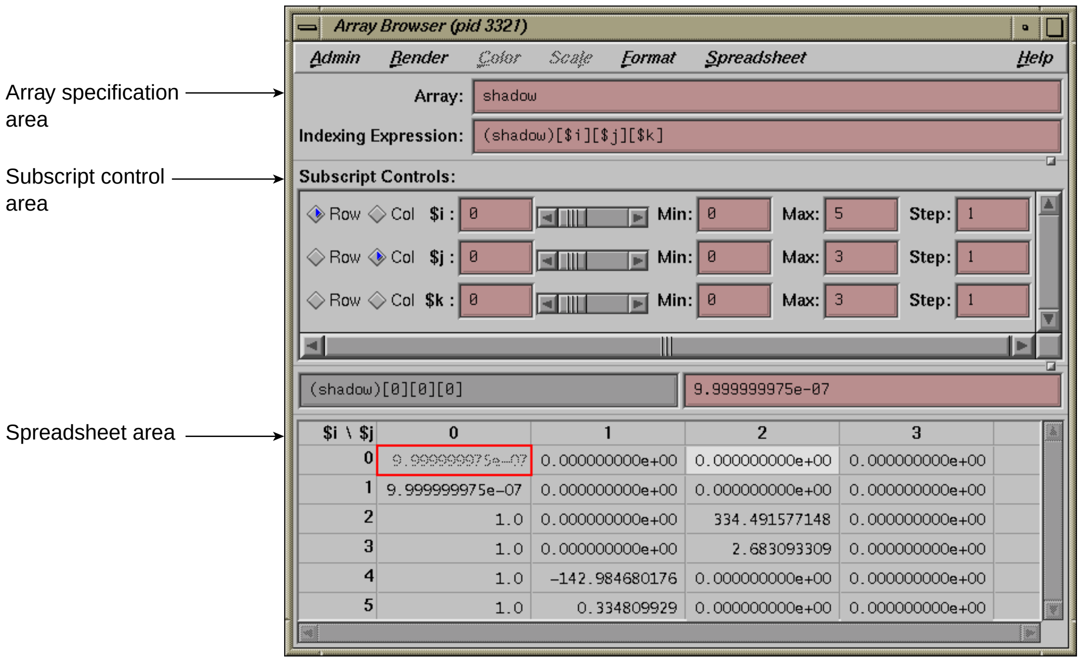
14. Select **Close** from the **Admin** menu of **Data Explorer** window to close it.

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 4-5, page 58, 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.



Array: shadow

Indexing Expression: (shadow)[$i][$j][$k]

◆ Row ◇ Col	$i :	0	Min: 0	Max: 5	Step: 1
◇ Row ◆ Col	$j :	0	Min: 0	Max: 3	Step: 1
◇ Row ◇ Col	$k :	0	Min: 0	Max: 3	Step: 1

(shadow)[0][0][0] 9.999999975e-07

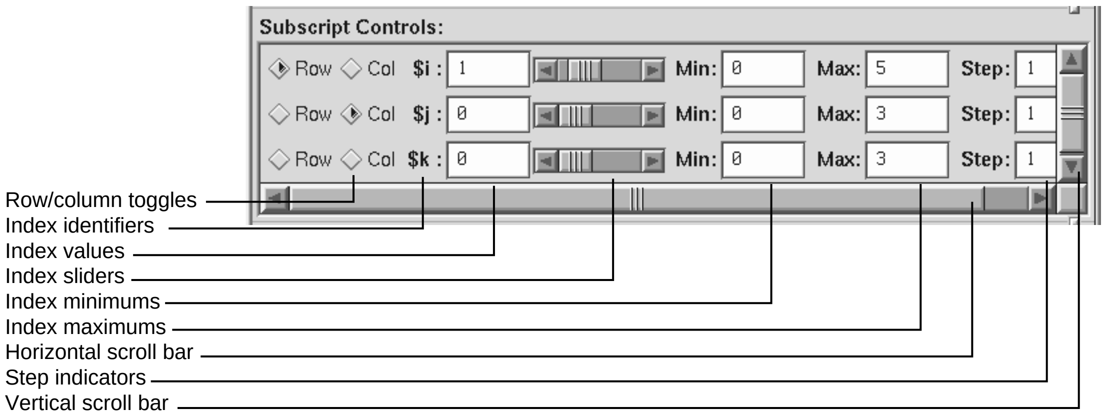
$i \ $j	0	1	2	3
0	9.999999975e-07	0.000000000e+00	0.000000000e+00	0.000000000e+00
1	9.999999975e-07	0.000000000e+00	0.000000000e+00	0.000000000e+00
2	1.0	0.000000000e+00	334.491577148	0.000000000e+00
3	1.0	0.000000000e+00	2.683093309	0.000000000e+00
4	1.0	-142.984680176	0.000000000e+00	0.000000000e+00
5	1.0	0.334809929	0.000000000e+00	0.000000000e+00

Figure 4-5 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 4-6, page 59, shows a close-up view of the subscript control area.


Subscript Controls:

◆ Row ◇ Col	$i :	1	Min:	0	Max: 5	Step: 1
◇ Row ◆ Col	$j :	0	Min:	0	Max: 3	Step: 1
◇ Row ◇ Col	$k :	0	Min:	0	Max: 3	Step: 1

Figure 4-6 `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.

Chapter 5

Setting Traps (Breakpoints)

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:

- "Traps Terminology ", page 61
- "Setting Traps", page 64
- "Setting Traps in the Trap Manager Window", page 66
- "Setting Traps by Using Signal Panel and System Call Panel", page 74

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 is 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 69 for trap syntaxes.

- A *stop trap* causes 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 only accept a “stop at” trap on a DSO (dynamic shared object) and line number loaded at startup. (See "Syntaxes", page 69 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 then remembers 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 also stop your program on throws of pointer, reference, const, and volatile types. For example, if you use the following, your program stops 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 64
- "Setting Traps Using the cvd Command Line", page 65
- "Setting Traps Using the Traps Menu in the Main View Window", page 65

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 to the left of the source code in the Main View or Source View windows. As you scroll the cursor up and down along the source code, the word “Stop” appears. Click the left mouse button to add or remove a trap. A menu is also available for setting and modifying the trap by holding down the right mouse button in the source annotation column (to the left of the source code).

  A subsequent click on the trap 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 67, 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.

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). 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 74.

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.

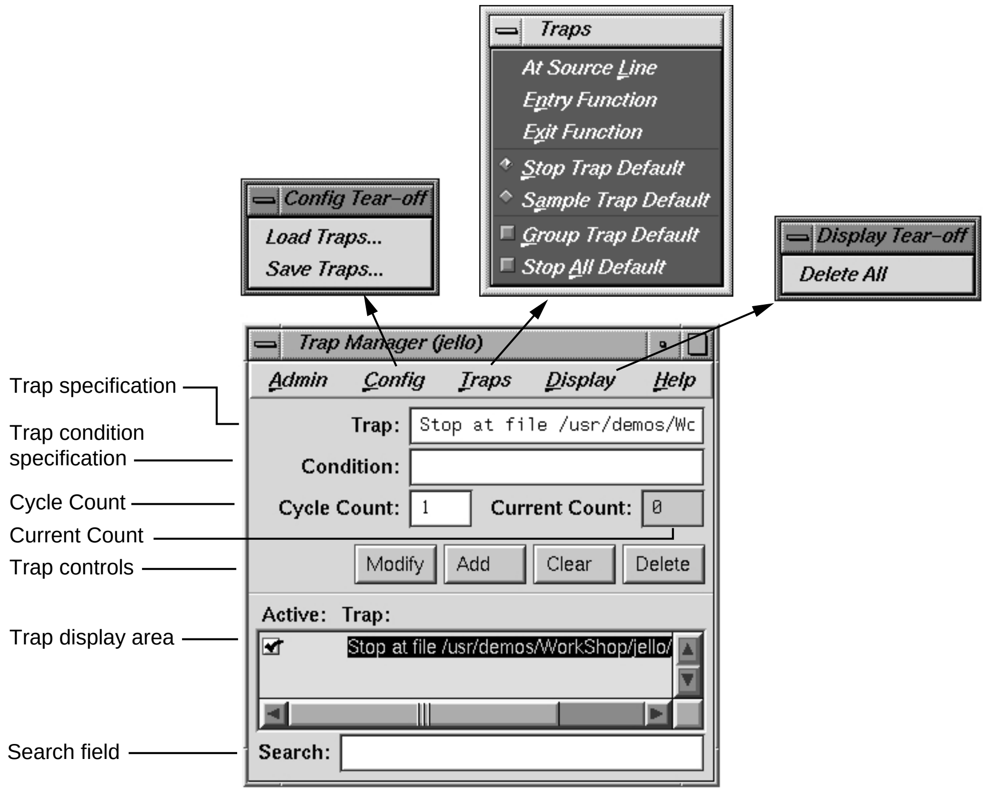
When the **Stop All Defaults** toggle is checked (ON), the `all` option is added into the resulting trap when a trap is set. This option causes the trap to apply to all processes/pthreads in the current debugging session.

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 major functions are to:

- List all traps in the process (except signal traps).
- Add, delete, modify, or disable the traps listed.
- Navigate the user code. Diving or double-clicking on a trap repositions the source in the Main View to the trap location.

The **Trap Manager** window appears in Figure 5-1 with the **Config**, **Traps**, and **Display** menus shown.



Trap: Stop at file /usr/demos/Wc
Condition:
Cycle Count: 1
Current Count: 0
Active: Trap:
Stop at file /usr/demos/WorkShop/jello/

Figure 5-1 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:

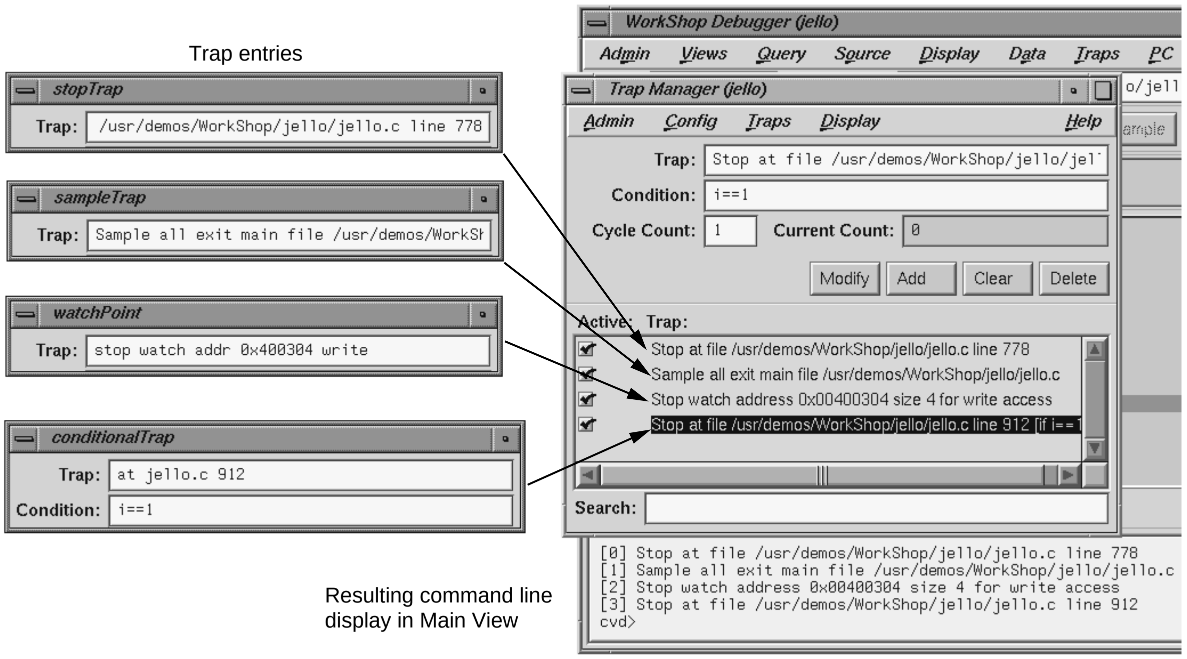
```
[stop|sample] [all] [pgrp] location | condition
```

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 5-2, page 68. 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.


Trap entries
stopTrap
Trap: /usr/demos/WorkShop/jello/jello.c line 778
sampleTrap
Trap: Sample all exit main file /usr/demos/WorkSh
watchPoint
Trap: stop watch addr 0x400304 write
conditionalTrap
Trap: at jello.c 912
Condition: i==1
WorkShop Debugger (jello)
Admin Views Query Source Display Data Traps PC
Trap Manager (jello)
Admin Config Traps Display Help
Trap: Stop at file /usr/demos/WorkShop/jello/jel
Condition: i==1
Cycle Count: 1 Current Count: 0
Modify Add Clear Delete
Active: Trap:
Stop at file /usr/demos/WorkShop/jello/jello.c line 778
Sample all exit main file /usr/demos/WorkShop/jello/jello.c
Stop watch address 0x00400304 size 4 for write access
Stop at file /usr/demos/WorkShop/jello/jello.c line 912 [if i==
Search:
Resulting command line display in Main View

```
[0] Stop at file /usr/demos/WorkShop/jello/jello.c line 778
[1] Sample all exit main file /usr/demos/WorkShop/jello/jello.c
[2] Stop watch address 0x00400304 size 4 for write access
[3] Stop at file /usr/demos/WorkShop/jello/jello.c line 912
cvd>
```

Figure 5-2 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 > Data Explorer**.

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 74.) 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 74.) 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 74.) 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 5-2, page 68. 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 fires 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

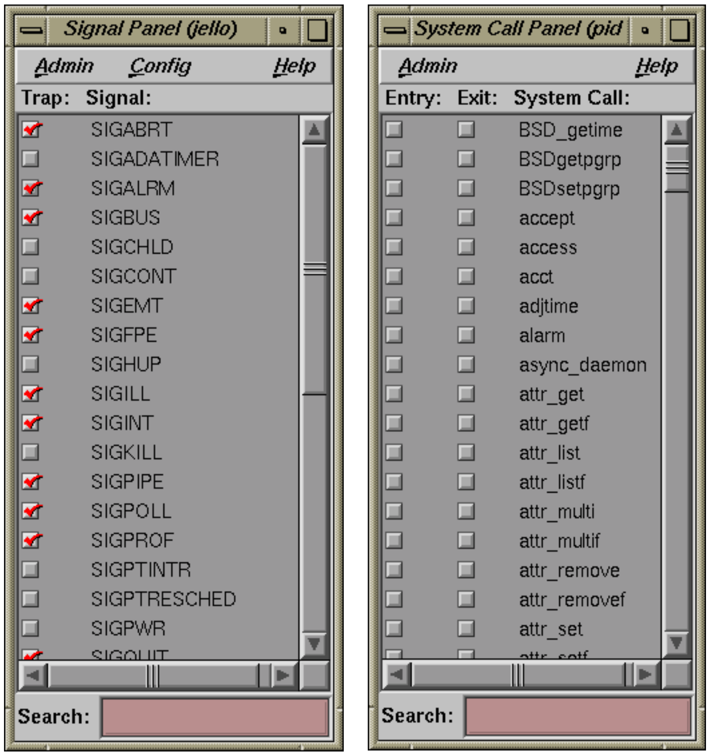
Each trap has an indicator to its left for toggling back and forth between active and inactive trap states. This feature lets you accumulate traps and turn them on only as needed. Thus, when you do not need the trap, it is not in your way. When you do need it, you can easily activate it.

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 5-3, page 75).



Trap:	Signal:
✓	SIGABRT
☐	SIGADATIMER
✓	SIGALRM
✓	SIGBUS
☐	SIGCHLD
☐	SIGCONT
✓	SIGEMT
✓	SIGFPE
☐	SIGHUP
✓	SIGILL
✓	SIGINT
☐	SIGKILL
✓	SIGPIPE
✓	SIGPOLL
✓	SIGPROF
☐	SIGPTINTR
☐	SIGPTRESCHED
☐	SIGPWR
✓	SIGQUIT

Entry:	Exit:	System Call:
☐	☐	BSD_getime
☐	☐	BSDgetpgrp
☐	☐	BSDsetpgrp
☐	☐	accept
☐	☐	access
☐	☐	acct
☐	☐	adjtime
☐	☐	alarm
☐	☐	async_daemon
☐	☐	attr_get
☐	☐	attr_getf
☐	☐	attr_list
☐	☐	attr_listf
☐	☐	attr_multi
☐	☐	attr_multif
☐	☐	attr_remove
☐	☐	attr_removef
☐	☐	attr_set
☐	☐	attr_setf

Figure 5-3 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.

Chapter 6

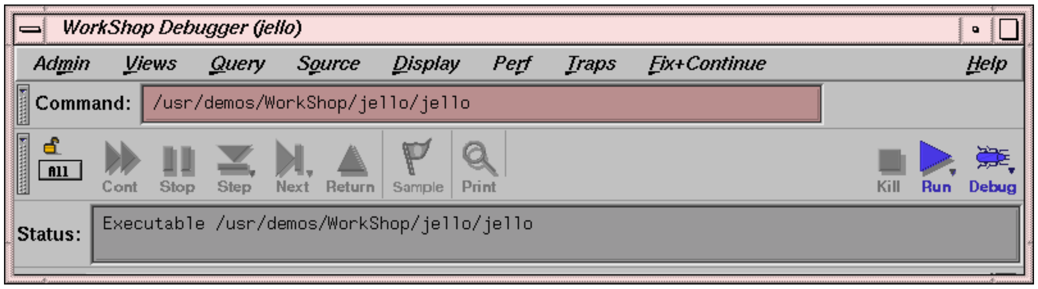
Controlling Program Execution

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", page 77
- "Controlling Program Execution Continue To/Jump To", page 79
- "Execution View", page 80

The Main View Window Control Panel

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



```
/usr/demos/WorkShop/jello/jello
```

```
Executable /usr/demos/WorkShop/jello/jello
```

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.
- Execution control buttons, which enable control of the program. These buttons are described in more detail in "Execution Control Buttons", page 78.

- **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 the *ProDev WorkShop: Debugger Reference Manual*.

- **Kill**: kills the active process.
- **Run**: creates a new process for your program and starts its execution. The **Run** button is also used to re-run a program.
- **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 is acted upon.
- The lock icon (**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 follows the interesting thread. This means it focuses on threads that reach a user breakpoint.
- **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** does 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 a pop-up menu. You can select one of the fixed values or enter your own number of steps by selecting

N. If you select **N**, a dialog box appears, allowing you to enter the number of instructions to step over.

- **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 Continue To/Jump To

The **PC** (program counter) menu is accessible by holding the mouse down over a line of code in the Source View window in the Main View window. The **Continue To** and **Jump To** menu picks 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 that you wish to target, then hold down the mouse button and select an item 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 takes 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:

- "Traceback Through the Call Stack Window", page 81
- "Options for Viewing Variables", page 82
- "Evaluating Expressions", page 84

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 the *ProDev WorkShop: Debugger Reference Manual*.

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.

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.

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 are 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 the *ProDev WorkShop: Debugger Reference Manual*.

Using Click to Evaluate

In the Main View window's source pane, a click of the right mouse button brings up a pop-up menu from which you can select **Click to Evaluate**. When this option is on, a click on a variable causes its value to appear. If you click on a subscript variable, its address appears. If you hold down the left mouse button and wipe across a variable and its subscripts to highlight them, the current value is 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 21 for short description of the Array Browser. The *ProDev WorkShop: Debugger Reference Manual* provides a more detailed description and graphic of the window and associated submenus.

Using the Data Explorer

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

This calls up the **Data Explorer** 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. The *ProDev WorkShop: Debugger Reference Manual* 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. The *ProDev WorkShop: Debugger Reference Manual* 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 displays 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. The *ProDev WorkShop: Debugger Reference Manual* section on the **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**.

Using the Data View Window

When you “dive” on a variable in the Source View window (click the right mouse button over a variable), the Data View window appears, displaying that variable’s information. Pointers and structures can be re-selected (re-dived) in the Data View window to expand portions of the object that is of interest.

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 the *ProDev WorkShop: Debugger Reference Manual* 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.

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	`? :`

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

Operation	Symbol
Member extraction[2]	`.  ->`
Assignment[3]	`= += -= /= %= >>= <<= &= ^= |=`
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, because 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 evaluates 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

2 These operations are interchangeable.

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

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 appears 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)	

Operation	Symbol
Function subroutine calls	
Assignment[4]	=

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.

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

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 is 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 is not executed when they combine. Modified functions are 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() is executed only when you rerun.

  – If you redefine `bar()` or `foobar()`, the new code is not executed when `foo()` returns. It is 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 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:

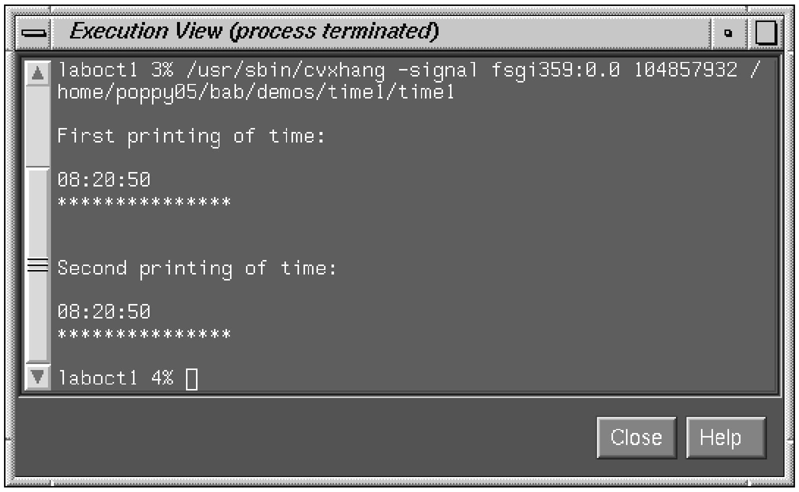
   ```
   % cd demos
   % mkdir time1
   % cd time1
   % cp /usr/demos/WorkShop/time1/* .
   % make time1
   % cvd time1 &
   ```

   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).



```
laboct1 3% /usr/sbin/cvxhang -signal fsgi359:0.0 104857932 /
home/poppy05/bab/demos/time1/time1

First printing of time:

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


Second printing of time:

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

laboct1 4%
```

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
   ***************
   ```

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
   ```

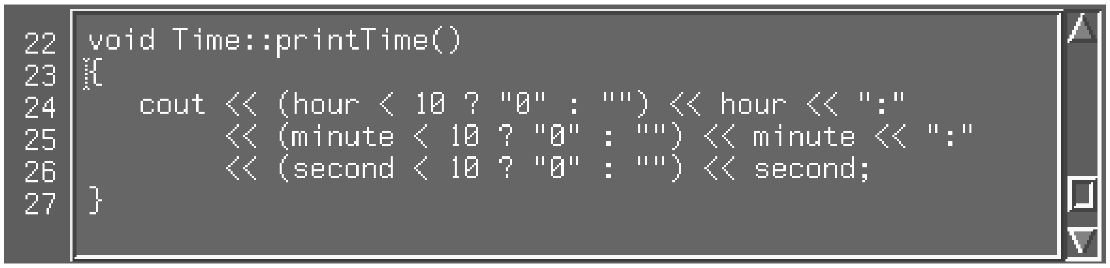
   This command opens the `time1/time1.c` file, which contains the implementation of class Time. The cursor is placed at the beginning of the `printTime` function. See Figure 8-2, page 97. The syntax of the `func` command is as follows:

   - For C++ class member functions:

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

   - For all other C/C++ functions:

     cvd> **func** ***functionName***

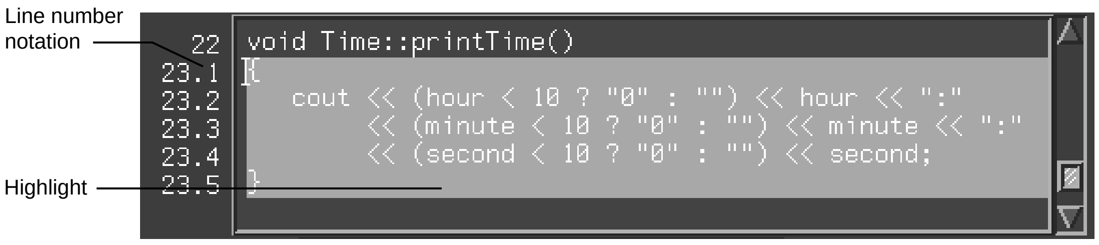


```
void Time::printTime()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

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 98. 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.



```
  22 void Time::printTime()
23.1 {
23.2    cout << (hour < 10 ? "0" : "") << hour << ":"
23.3         << (minute < 10 ? "0" : "") << minute << ":"
23.4         << (second < 10 ? "0" : "") << second;
23.5 }
```

Figure 8-3 Redefined Function

Changing Code

1. To change the time output as shown in Step 5 in "Editing a Function", page 96, 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 98. 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 use this ID in the procedure in "Switching between Compiled and Redefined Code", page 103. 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 the *ProDev WorkShop: Debugger Reference Manual*.

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.

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.

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
   ```

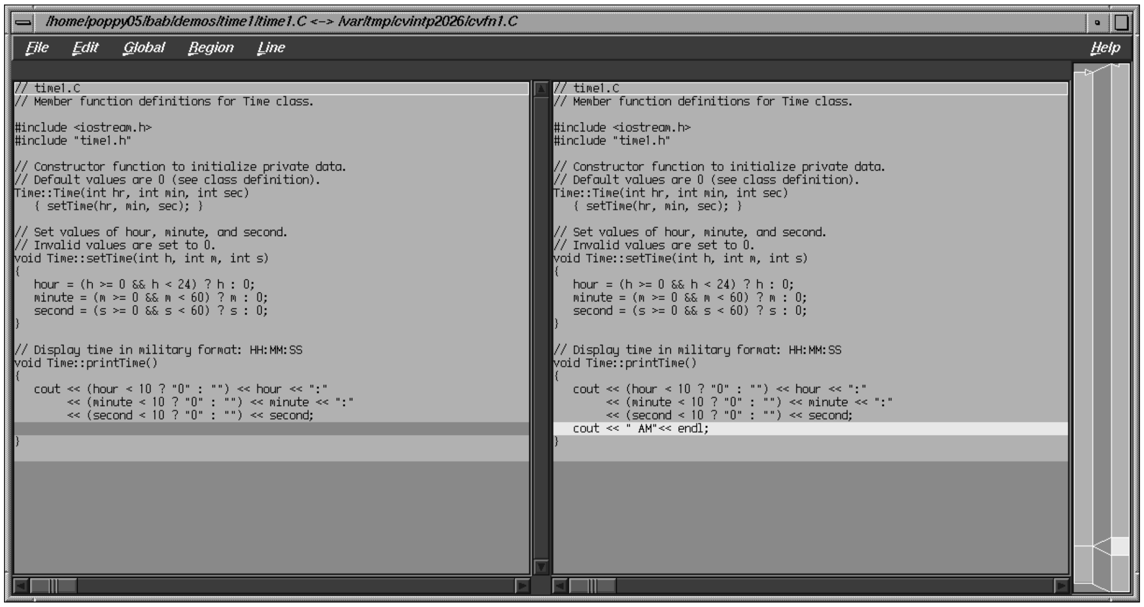
7. Select **Fix+Continue > Edited<–>Compiled** from the menu bar to re-enable your changes.
8. Click on the **Run** button to see the changed printout of `Time`. The following displays:

   ```
   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:


/home/poppy05/bab/demos/time1/time1.C <-> /var/tmp/cvintp2026/cvfn1.C
File
Edit
Global
Region
Line
Help

```
// time1.C
// Member function definitions for Time class.

#include <iostream.h>
#include "time1.h"

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
   { setTime(hr, min, sec); }

// Set values of hour, minute, and second.
// Invalid values are set to 0.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Display time in military format: HH:MM:SS
void Time::printTime()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;

}
```

```
// time1.C
// Member function definitions for Time class.

#include <iostream.h>
#include "time1.h"

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time(int hr, int min, int sec)
   { setTime(hr, min, sec); }

// Set values of hour, minute, and second.
// Invalid values are set to 0.
void Time::setTime(int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
   minute = (m >= 0 && m < 60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
}

// Display time in military format: HH:MM:SS
void Time::printTime()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
   cout << " AM"<< endl;
}
```

Figure 8-4 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", page 107
- "Finding Heap Corruption Errors", page 107
- "Heap Corruption Detection Tutorial", page 111

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
   ```

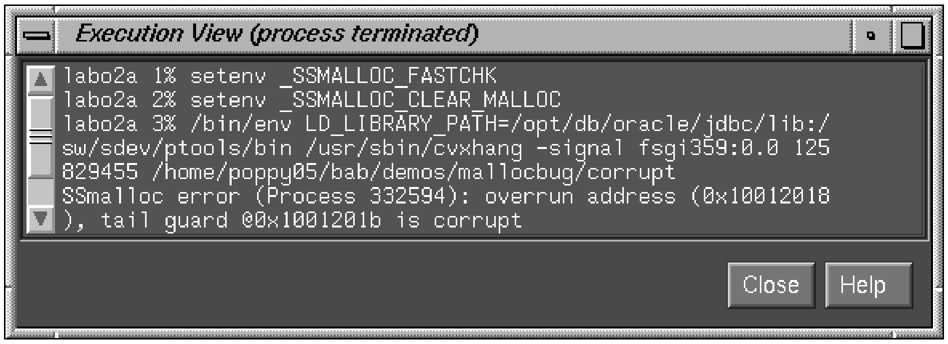
   A stop trap is set at the exit from the `malloc` library `ssmalloc_error`.

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 113.)

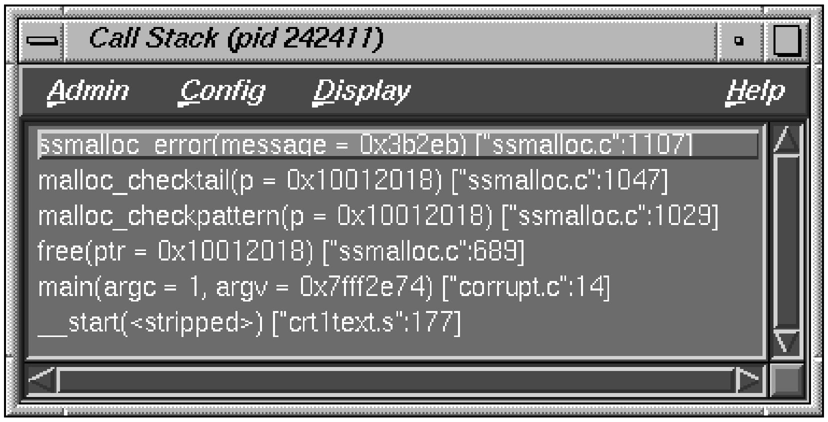


```
labo2a 1% setenv _SSMALLOC_FASTCHK
labo2a 2% setenv _SSMALLOC_CLEAR_MALLOC
labo2a 3% /bin/env LD_LIBRARY_PATH=/opt/db/oracle/jdbc/lib:/
sw/sdev/ptools/bin /usr/sbin/cvxhang -signal fsgi359:0.0 125
829455 /home/poppy05/bab/demos/mallocbug/corrupt
SSmalloc error (Process 332594): overrun address (0x10012018
), tail guard @0x1001201b is corrupt
```

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).



```
ssmalloc_error(message = 0x3b2eb) ["ssmalloc.c":1107]
malloc_checktail(p = 0x10012018) ["ssmalloc.c":1047]
malloc_checkpattern(p = 0x10012018) ["ssmalloc.c":1029]
free(ptr = 0x10012018) ["ssmalloc.c":689]
main(argc = 1, argv = 0x7fff2e74) ["corrupt.c":14]
__start(<stripped>) ["crt1text.s":177]
```

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.

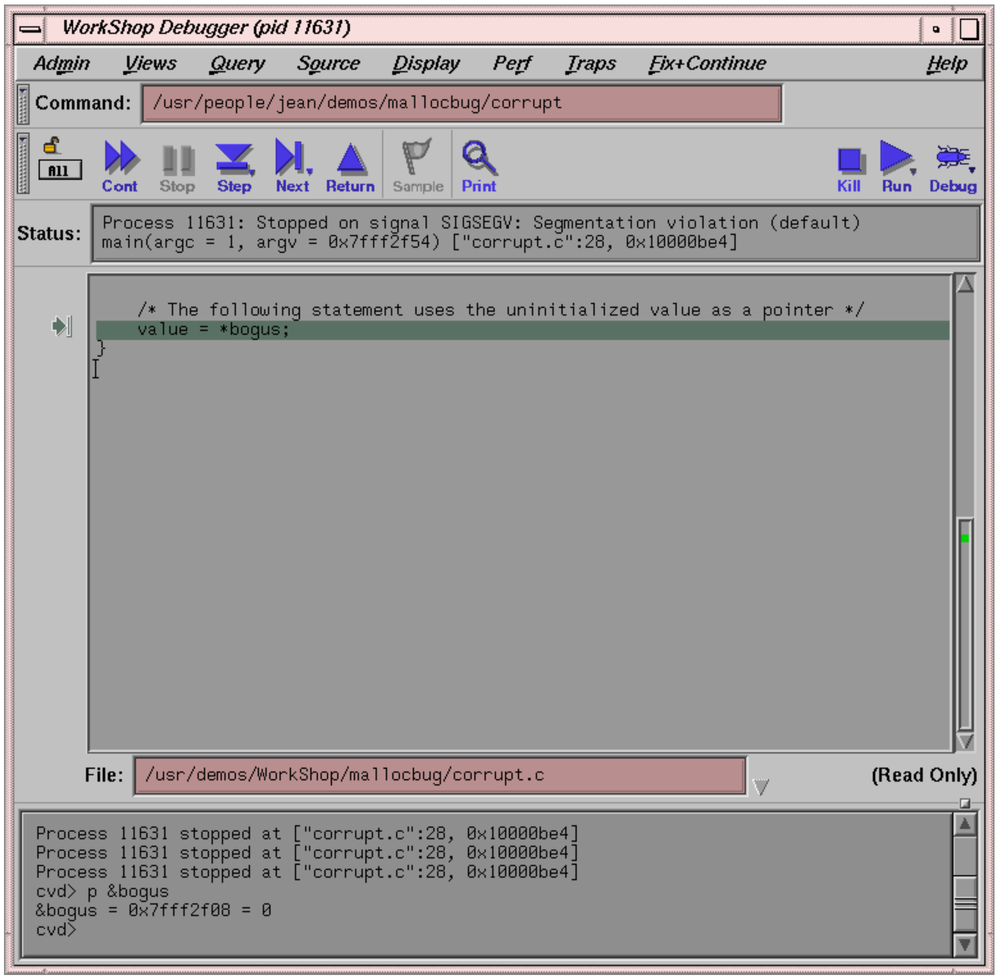
8. Click the **Continue** button again and watch the **Execution View** and **Call Stack** windows.

   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 115. We need the bad address so that we can set a watchpoint to find out when it is written to.



```
/usr/people/jean/demos/mallocbug/corrupt
```

```
Process 11631: Stopped on signal SIGSEGV: Segmentation violation (default)
main(argc = 1, argv = 0x7fff2f54) ["corrupt.c":28, 0x10000be4]
```

```
    /* The following statement uses the uninitialized value as a pointer */
    value = *bogus;
}
```

```
/usr/demos/WorkShop/mallocbug/corrupt.c
```

```
Process 11631 stopped at ["corrupt.c":28, 0x10000be4]
Process 11631 stopped at ["corrupt.c":28, 0x10000be4]
Process 11631 stopped at ["corrupt.c":28, 0x10000be4]
cvd> p &bogus
&bogus = 0x7fff2f08 = 0
cvd>
```

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 115). 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.

Chapter 10

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 Explorer** 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 the *ProDev WorkShop: Debugger Reference Manual* for a description of that menu).

Currently, **Multiprocess Explorer** 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 `apo(5)` man page.
- *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:

- "Using the Multiprocess Explorer Window", page 118
- "Debugging a Multiprocess C Program", page 122
- "Debugging a Multiprocess Fortran Program", page 128

- "Debugging a Pthreaded Program", page 133
- "Debugging an MPI Single System Image Application", page 144
- "Debugging an OpenMP Application", page 152

Using the Multiprocess Explorer Window

The **Multiprocess Explorer** window is brought up by selecting **Admin > Multiprocess Explorer** 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 does not become part of the process group.

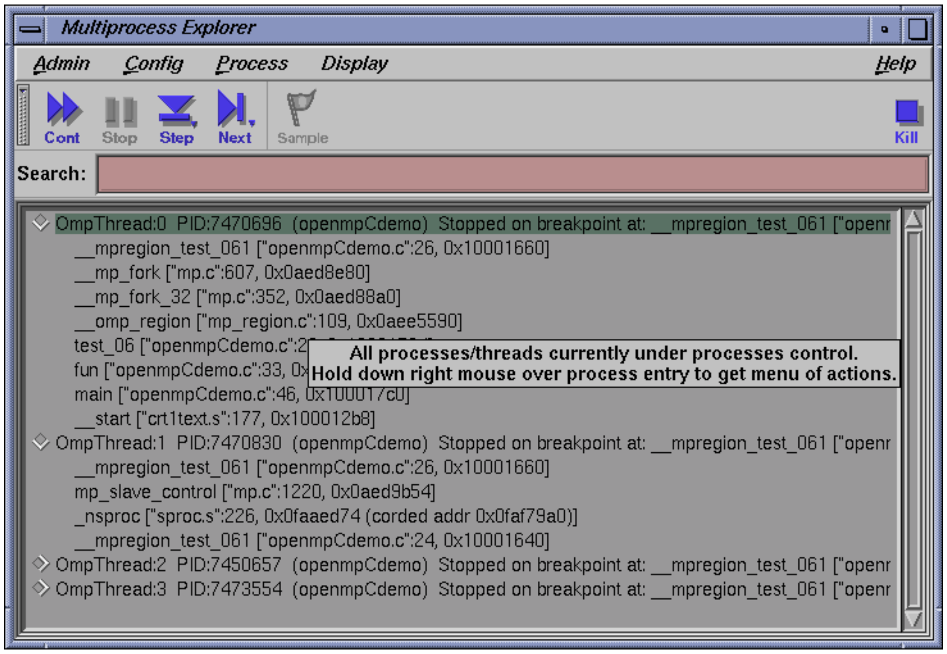
Each process in the session can have a standard main view window session associated with it. However, all processes in a process group appear on a single **Multiprocess Explorer** window.

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 Explorer** from the menu bar.

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



```
OmpThread:0  PID:7470696  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
    __mpregion_test_061 ["openmpCdemo.c":26, 0x10001660]
    __mp_fork ["mp.c":607, 0x0aed8e80]
    __mp_fork_32 ["mp.c":352, 0x0aed88a0]
    __omp_region ["mp_region.c":109, 0x0aee5590]
    test_06 ["openmpCdemo.c":2
    fun ["openmpCdemo.c":33, 0x
    main ["openmpCdemo.c":46, 0x100017c0]
    __start ["crt1text.s":177, 0x100012b8]
OmpThread:1  PID:7470830  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
    __mpregion_test_061 ["openmpCdemo.c":26, 0x10001660]
    mp_slave_control ["mp.c":1220, 0x0aed9b54]
    _nsproc ["sproc.s":226, 0x0faaed74 (corded addr 0x0faf79a0)]
    __mpregion_test_061 ["openmpCdemo.c":24, 0x10001640]
OmpThread:2  PID:7450657  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
OmpThread:3  PID:7473554  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
```

All processes/threads currently under processes control.
Hold down right mouse over process entry to get menu of actions.

Figure 10-1 **Multiprocess Explorer**

Viewing Process Status

The process display area of the **Multiprocess Explorer** lists the status of all processes and threads in the process group. For definitions of the various statuses and states, see the *ProDev WorkShop: Debugger Reference Manual*.

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 Main View focus to a different process or thread
- create a Main View window for a different process and/or thread
- focus Main View attention to a user-entered thread
- show process or thread-specific details
- add or remove a process entry

For complete details about the **Process** menu, see the *ProDev WorkShop: Debugger Reference Manual*.

Using Multiprocess Explorer Control Buttons

The **Multiprocess Explorer** 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 Explorer** 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 the *ProDev WorkShop: Debugger Reference Manual*.

Multiprocess Traps

As discussed in Chapter 5, "Setting Traps (Breakpoints)", page 61, 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 Multiprocess and Pthreaded Applications

The **Multiprocess Explorer** 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 Explorer** window to get additional information about a process or thread:

In **Display > Process** mode:

1. Double-click on a folder icon.

   The process display expands to show its pthreads, if any. If there are no pthreads, the call stack for the process is displayed if the process is displayed.

2. Double-click to select the pthread of your choosing.

   The call stack for that pthread displays if the pthread is stopped.

In **Display > Status** mode:

1. Double-click on a folder icon.

   The status display expands to show a list of processes.

2. Double-click to select the process of your choosing.

   The process display expands to show its pthreads if any. If there are no pthreads, the call stack for the process is displayed if the process is stopped.

3. Double-click to select the pthread of your choosing.

   The call stack for the selected pthread is displayed if the pthread is stopped.

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 Explorer** 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 Explorer Preferences** dialog. The preferences on this dialog let 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 Explorer Preference** options, see the *ProDev WorkShop: Debugger Reference Manual*.

Bringing up Additional Main View Windows

To create a Main View window for a process, highlight that process in the **Multiprocess Explorer** window. Then, select **Process > Create new window** in the **Multiprocess Explorer** window. Starting with WorkShop 2.9.2, the user can "dive" via the mouse button on entries in the **Multiprocess Explorer** window. In the above example, the right mouse button can be held over the process selection and a dynamic process menu is displayed. You can then select **Create new 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 Explorer** 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 Explorer

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 Explorer** by selecting **Admin > Multiprocess Explorer** 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 Explorer 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:

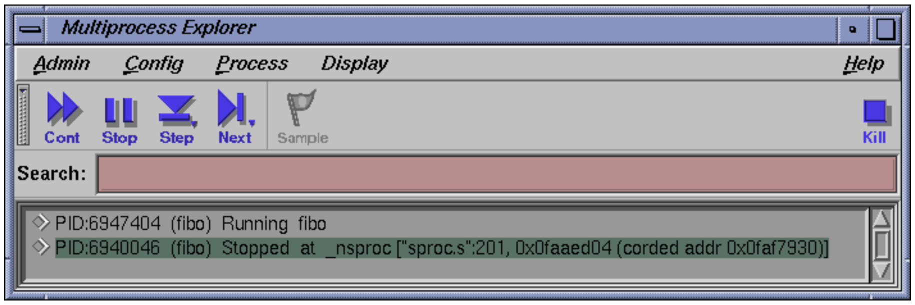
1. Select **Config > Preferences** from the menu bar in the **Multiprocess Explorer** window.

2. Toggle off **Resume child after attach on sproc** in the **Multiprocess Explorer 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 Explorer** 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 Explorer** window.

   The control buttons on the **Multiprocess Explorer** 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 Explorer** window to highlight this line.



```
PID:6947404 (fibo) Running fibo
PID:6940046 (fibo) Stopped at _nsproc ["sproc.s":201, 0x0faaed04 (corded addr 0x0faf7930)]
```

Figure 10-2 Multiprocess Explorer with highlighted process

8. Select **Process > Create new window** from the menu bar of the **Multiprocess Explorer** 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 does 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 Explorer** window. The first child, created in Step 5, now spawns a grandchild process that stops in `_nsproc`.
13. A Main View window switches to the new stopped process. Click on **Stop** in the **Multiprocess Explorer** 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 Explorer to Control Execution", page 124.

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 (Breakpoints)", page 61.

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 affects only the parent process.

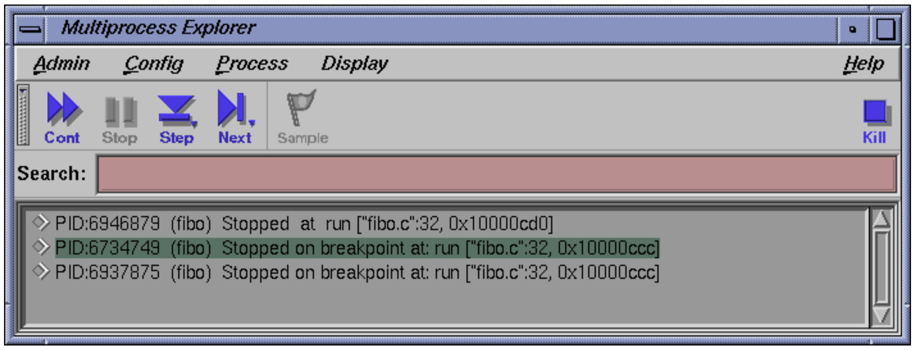
4. Click on the **Cont** button in the **Multiprocess Explorer** window. The parent process has stopped, but the other processes are probably still running.

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 Explorer** window to see the running processes in the process group stop at the trap on line 32.



```
PID:6946879  (fibo)  Stopped  at  run ["fibo.c":32, 0x10000cd0]
PID:6734749  (fibo)  Stopped on breakpoint at: run ["fibo.c":32, 0x10000ccc]
PID:6937875  (fibo)  Stopped on breakpoint at: run ["fibo.c":32, 0x10000ccc]
```

Figure 10-3 Multiprocess Explorer with running processes stopped

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 causes 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` is also 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 Explorer Preferences** dialog. To open this dialog, select **> Admin > Multiprocess Explorer** from the Main View menu bar to open the **Multiprocess Explorer** window.

    Then, select **Config > Preferences** from within the **Multiprocess Explorer** 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:

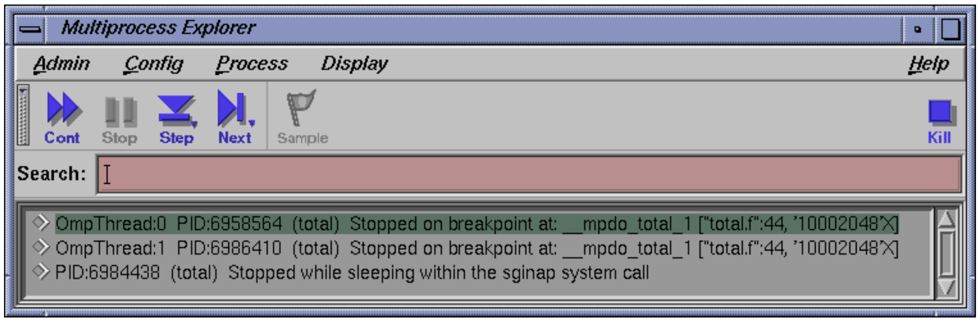
    ```
    Stop pgrp at file /usr/demos/WorkShop/mp/total.f line 44
    ```

13. Select **Admin > Multiprocess Explorer** 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 Explorer**, 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 if it has switched to the slave process.



```
OmpThread:0  PID:6958564  (total)  Stopped on breakpoint at: __mpdo_total_1 ["total.f":44, 10002048X]
OmpThread:1  PID:6986410  (total)  Stopped on breakpoint at: __mpdo_total_1 ["total.f":44, 10002048X]
PID:6984438  (total)  Stopped while sleeping within the sginap system call
```

Figure 10-4 Multiprocess Explorer: stopped at breakpoint

15. Right-click on the name of the slave process in the **Multiprocess Explorer** 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 the values for `ii` and `jj`. 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

Using the Debugger you can view pthread creation and execution through the **Multiprocess Explorer** 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 Explorer** window when debugging pthreads. Ada users should use the **Task View** window.

The next sections give hints on debugging pthreaded programs and illustrate how to debug a program that uses IRIX 6.5 pthreads.

User-Level Preferences

As of the WorkShop 2.9.3 release, two user-level preferences are available through the command-line view (**DbxView** in **cvd**). These preferences control the setting of one-time, internal breakpoints that are used to create deterministic behavior when native pthreads experience 'blocking' (described in the following subsections).

If these breakpoints are not set, the user can experience a 'Running' display in the debugger, with the appearance that the debugger is stuck in a loop.

These breakpoints are only set internally and only when a single, native pthread is being ’stepped’. Otherwise the breakpoints do not appear.

The preferences that can be set from the command line view or from the `dbx` command line are as follows:

- `set $pthreadSchedBlockBkpt = true`: This causes a one-time, internal breakpoint to be set in the `__SGIPT_sched_block` pthread library routine when a single native pthread is ’stepped’. If the single thread ’blocks’ in one of the blocking pthread library routines (mainly in the `mutex` library routines) , the thread stops on a breakpoint and the user must decide the next action (for example, continue a different thread, or continue all threads). The default is `true`. See "Blocking pthreads Library Routines", page 144, for information about blocking kernel library routines.
- `set $pthreadSyscallBlockBpt = true`: This causes a one-time, internal breakpoint to be set in the `__SGIPT_libc_blocking` pthread library routine when a single native pthread is ’stepped’. If the single thread ’blocks’ in one of the ’blocking’ kernel system calls (for example, `_writev`, which underlines `printf`), the thread stops on a breakpoint and the user must decide on the next action (for example, continue a different thread or continue all threads). The default is `false`. See the *ProDev WorkShop: Debugger Reference Manual* for a complete list of blocking kernel system calls.

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 sees 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, 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 the *ProDev WorkShop: Debugger Reference Manual* 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 Explorer** 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 Explorer** not **Task View**.
- Use the WorkShop tools instead of `dbx` for 6.5 pthread debugging whenever possible.
- Do not do a **Next** of `printf`.

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. However, a pthread example is provided in this section.

See "User-Level Continue of Single 6.5 POSIX Pthread", page 134, for an in-depth description of current pthread implementation in IRIX.

pthread example

The following is a sample program used in this pthread example:

```
#################### begin highlight program##########################
#include        <unistd.h>

#define PTMAX 4
#define ERR(t,m) if (t) {printf("%s\n",m); exit(1);}

pthread_t phandle[PTMAX];
int     arg[PTMAX];

int foo( int *threadnum )
{
  int num, val = 0;

  num = (*threadnum) & 0x0000000F;
  printf("enter foo ( 0x%08x)\n", *threadnum);

  switch ( num ) {
    case  1: val = foo1(*threadnum);  break;
    case  2: val = foo2(*threadnum);  break;
    case  3: val = foo3(*threadnum);  break;
    case  4: val = foo4(*threadnum);  break;
    default: {printf("ERROR: foo(%d)\n",num); break;}
  }

  return val;
}

int foo1(int threadnum)
{
  return threadnum;
}

int foo2(int threadnum)
{
  return threadnum;
}

int foo3(int threadnum)
{
  int l0,j0;
  l0 = threadnum;
  j0 = l0+threadnum;
  return j0;
}

int foo4(int threadnum)
{
  return threadnum;
}

void create(int threadnum)
{
  int stat;

  arg[threadnum]=threadnum+1;
  printf("create: threadnum=0x%08x\n",arg[threadnum]);
  stat= pthread_create(&phandle[threadnum],0,(void *(*)(void *))&foo,&arg[threadnum]);

  ERR(stat!=0,"pthread_create failed");
}

void join(int threadnum)
{
  int out, stat;

  printf("join  : threadnum=0x%08x\n",threadnum+1);
  stat= pthread_join(phandle[threadnum],(void **)&out);
  ERR(stat!=0,"pthread_join failed");
  printf("return: threadnum=0x%08x: out=0x%08x\n",threadnum+1,out);
}

int main( int argc, char **argv )
{
  int threadnum;

  for( threadnum=0; threadnum<PTMAX; threadnum++ ) {
    create(threadnum);
    sleep(1);
  }

  for( threadnum=0; threadnum<PTMAX; threadnum++ ) {
    join(threadnum);
  }
}
#################### end highlight program##########################
```

The `pthreaddemo` is a simple program that creates 4 pthreads. Each pthread is created via the `pthread_create()` routine, which in turn calls its start routine. Then pthread 0x10001 calls `foo1()`, pthread 0x10002 calls `foo2()`, pthread 0x10003 calls `foo3()`, and finally pthread 0x10004 calls `foo4()`. The master will sleep 1 second after each call just to make this example more predictable. As each routine encounters a breakpoint, control is given back to the user.

Perform the following to stat, compile the program, and run the Debugger:

1. Copy the program source from the `demo` directory as follows:

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

2. Compile `pthreaddemo.c` by entering the following command:

   ```
   % cc -g -o pthreaddemo pthreaddemo.c -lpthread -lc
   ```

3. Invoke the Debugger on `pthreaddemo` as follows:

   ```
   % cvd pthreaddemo
   ```

4. Invoke the **Multiprocess Explorer** by selecting **Admin > Multiprocess Explorer** from the Main View menu bar.

   The next section uses the `pthreaddemo` program to illustrate some of the functionality of the Multiprocess window when using 6.5 pthreads.

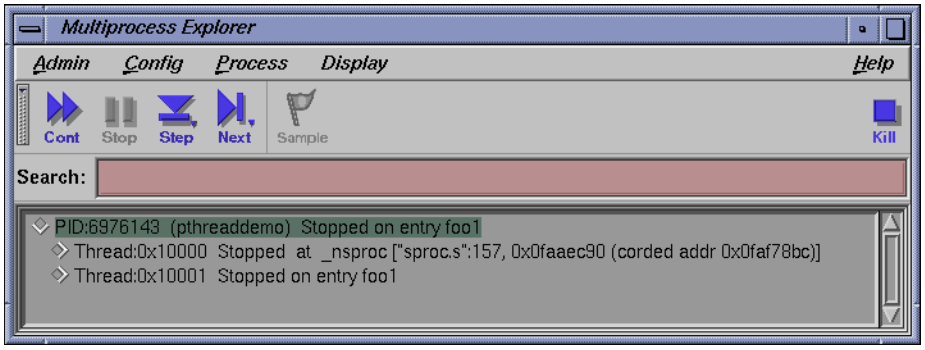
5. At the `cvd` prompt in the Main View window, enter the following to set breakpoints:

   ```
   cvd> stop in foo1
   cvd> stop in foo2
   cvd> stop in foo3
   cvd> stop in foo4
   ```

   This sets a breakpoint in a unique routine that will be called by each pthread.

6. Click the **Run** button in the Debugger Main View window to run the program. The breakpoint in pthread 0x10001 should stop in `foo1()`.

7. In the Main View window, you should be able to confirm this by noting the program counter (PC) being highlighted in the source. Also, in the Multiprocess Explorer you should see the following:



```
PID:6976143  (pthreaddemo)  Stopped on entry foo1
  Thread:0x10000  Stopped  at  _nsproc ["sproc.s":157, 0x0faaec90 (corded addr 0x0faf78bc)]
  Thread:0x10001  Stopped on entry foo1
```

Figure 10-5 Pthread stopped on entry

8. At the `cvd` command prompt at the bottom of the Main View window, enter the following command:

   ```
   cvd> print threadnum
   thread = 1
   ```

   It should be the same number as the pthread you are focused on (for example, 1 for 0x10001 at this breakpoint.)

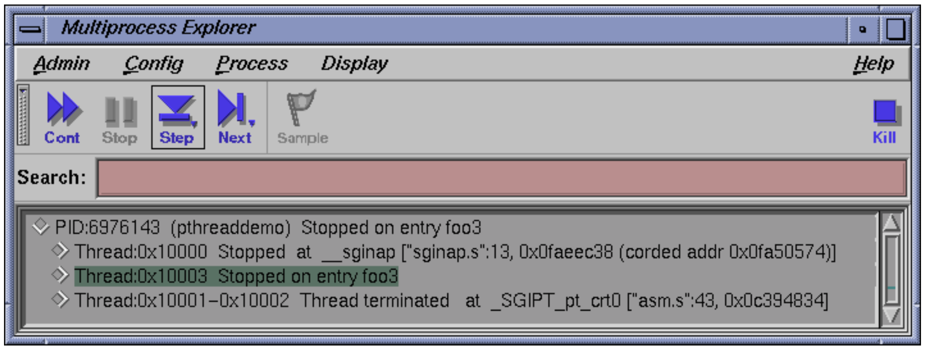
9. Click the **Cont** button in the Main View window. The second breakpoint, the one set in pthread 0x10002, should stop in `foo2()`.

10. In the Main View window, you should be able to confirm this by noting the green program counter (PC) being highlighted in the source. Also, in the Multiprocess Explorer you should see:

    ```
    Thread:0x10002 Stopped on entry foo2
    ```

    The Multiprocess Explorer should confirm this location for pthread 0x10002.

11. Click the **Cont** button in the Multiprocess Explorer. The third breakpoint, the one set in pthread 0x10003, should stop in `foo3()`. Both the Main View window and the Multiprocess Explorer window should confirm this location for pthread 0x10003.



```
PID:6976143 (pthreaddemo) Stopped on entry foo3
  Thread:0x10000 Stopped at __sginap ["sginap.s":13, 0x0faeec38 (corded addr 0x0fa50574)]
  Thread:0x10003 Stopped on entry foo3
  Thread:0x10001-0x10002 Thread terminated at _SGIPT_pt_crt0 ["asm.s":43, 0x0c394834]
```

Figure 10-6 Pthread stopped on entry 3

12. In the Debugger Main View window toggle the button below the **lock** to **Single**. Toggling this button to **Single** ensures the Debugger commands (`cont`, `next`, etc.) will only pertain to the **Single** pthread currently focused on in the Main View window.

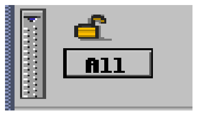


Figure 10-7 "All" toggle button

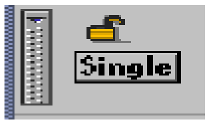


Figure 10-8 "Single" toggle button

13. Click on the **Next** button in the Main View window. Only pthread 0x10003 should advance one source line.

14. Clicking on the **Return** button in the Main View window should return pthread 0x10003 to its calling function.

15. Click the **Cont** button in the Multiprocess Explorer. The Main View window should stop at the fourth and final breakpoint in `foo4()`.

16. A final click on the **Cont** button in the Mulitprocess Explorer should continue to completion.

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 144)
- One of several blocking kernel syscalls (see the *ProDev WorkShop: Debugger Reference Manual* 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 the *ProDev WorkShop: Debugger Reference Manual* 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

The Debugger 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 `-np 2` *MPI_app_name* arguments are passed to `mpirun` and that `cvd` is initially focused on the `mpirun` process.

An entry point into the MPI application can be used to set a pending trap (breakpoint) in the MPI application. This breakpoint is resolved when the **Run** button is activated and the actual MPI application is running. If the breakpoint target is valid, the MPI application stops at the breakpoint and further debugging can be done.

Note: As of the WorkShop 2.9.2 release, `cvd` stops first in a special breakpoint in the `mpirun` command. If you have set a pending breakpoint in the actual MPI application, use the **Multiprocess Explorer continue** button to reach the breakpoint.

The use of the **Multiprocess Explorer** for debugging MPI applications is very similar to presentations in the previous and following 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 Explorer** window.

MPI Debugging Session

MPI debugging is highly variable, not only from environment to environment but also from IRIX release to IRIX release and the version of MPT installed on the system. Because of this, it is not possible to provide a representative MPI debugging tutorial that can be used by all users. However, an MPI example is provided in this section.

The following sample program is used in this pthread example (`/usr/demos/WorkShop/mp/mpidemo.c`):

```
#include <alloca.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#if __linux
#include <malloc.h>
#endif

#define ALIGN   16384

main(int argc, char **argv)
{
    MPI_Status status;
    int i, len, num, tag, size, rank, peer;
    double s, t, min, max, ave, *vec;
    char *stmp, *rtmp;
    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    len = 0;
    num = 10000;

    if (argc > 1) len = atoi(argv[1]);
    if (argc > 2) num = atoi(argv[2]);


    vec = alloca(num * sizeof(double));

    stmp = memalign(ALIGN, len + 163840);
    assert(stmp);

    rtmp = stmp + 128;

    tag = 0;

    switch (rank) {
     case 0:
            peer = 1;

            MPI_Send(stmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD);
            MPI_Recv(rtmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD, &status);
            t = MPI_Wtime();

            s = MPI_Wtime();

            for (i=0; i<num; i++) {
                MPI_Send(stmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD);
                MPI_Recv(rtmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD, &status);

                t = MPI_Wtime();

                vec[i] = t - s;

                s = t;
            }
            min = 100000000.0;
            max = 0.0;
            ave = 0.0;

            for (i=0; i<num; i++) {
               t = vec[i];

               if (min > t) min = t;
               if (max < t) max = t;

               ave += t;
            }

            if (num) {
               ave /= num;

              printf("%d bytes @ %d reps  best: %f MB/s  %f us  average: %f MB/s  %f us\n",
                   len, num, 2.0e-6*len/min, 0.5e6*min, 2.0e-6*len/ave, 0.5e6*ave);
            }

            break;

     case 1:
            peer = 0;

           MPI_Recv(rtmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD, &status);
           MPI_Send(stmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD);

           for (i=0; i<num; i++) {
                MPI_Recv(rtmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD, &status);
                MPI_Send(stmp, len, MPI_BYTE, peer, tag++, MPI_COMM_WORLD);
           }

           break;

     default:
           break;
    }

    while (num == 0) ;
    MPI_Finalize();
    return 0;
}
```

Perform the following commands to stat, compile the program, and run the Debugger. MPI must be installed on the system to compile and run this program. Array services must be running to run the actual MPI application:

1. Copy the program source from the demo directory as follows:

   ```
   % cp /usr/demos/WorkShop/mp/mpidemo.c .
   ```

2. Compile `mpidemo.c` by entering the following command:

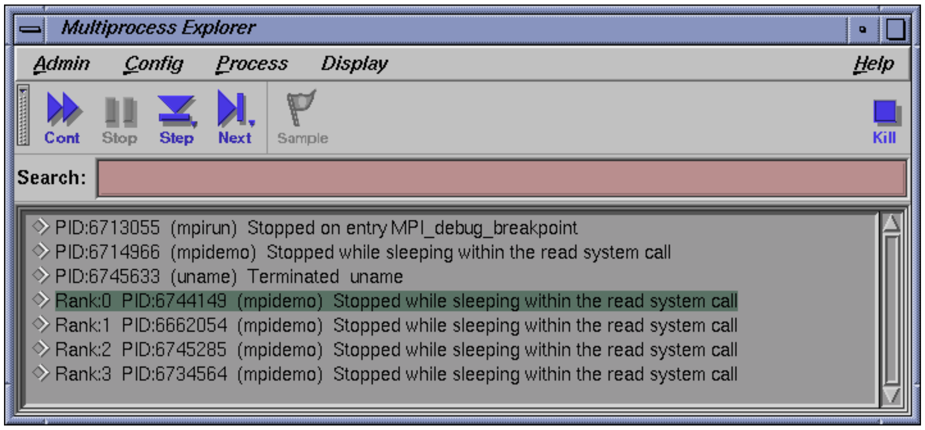
   ```
   % cc -g -o mpidemo mpidemo.c -lmpi -lc
   ```

3. Load the `mpirun` command into `cvd` with the `mpidemo` executable and 4 processors as the arguments of to `mpirun` (see the `mpirun` man page for more information about `mpirun` arguments).

   ```
   % cvd mpirun -args -np 4 mpidemo
   cvd> run
   ```

   This executes the command `mpirun -np 4 mpidemo` under `cvd` control.

4. The **MultiProcess Explorer** should stop with the following output when **Display by Process** mode is selected (PID values will differ from run to run):



```
PID:6713055  (mpirun)  Stopped on entry MPI_debug_breakpoint
PID:6714966  (mpidemo)  Stopped while sleeping within the read system call
PID:6745633  (uname)  Terminated  uname
Rank:0  PID:6744149  (mpidemo)  Stopped while sleeping within the read system call
Rank:1  PID:6662054  (mpidemo)  Stopped while sleeping within the read system call
Rank:2  PID:6745285  (mpidemo)  Stopped while sleeping within the read system call
Rank:3  PID:6734564  (mpidemo)  Stopped while sleeping within the read system call
```

Figure 10-9 Multiprocess Explorer: Display by Process

The `mpirun` command is now stopped at a special breakpoint for debuggers.

The process running `mpidemo` with no Rank value (PID:167060 in this example) is an MPI daemon process that controls the Rank children which are running the actual `mpidemo` MPI program. In some cases after the initial `run` command from `cvd`, `cvd` may stop this process in `_fork`. Use the **Multiprocess Explorer continue** button to continue the launching of the `mpidemo` program. After the `mpidemo` MPI rank children appear in the **Multiprocess Explorer** display, the **Multiprocess Explorer** focuses the Main View on the MPI rank 0 process and stops.

Depending on what shell is running, other processes that `mpirun` used to launch `mpidemo` may appear. In this example using `ksh`, the `uname` process shows up as a terminated process. You can ignore any terminated processes, the MPI daemon process, and the `mpirun` command.

The real MPI processes of interest are the **MPI rank children processes**. Those are displayed with the string *Rank:N* prior to the PID string. *N* is an integer from 0 to number of MPI processes -1. The number of MPI processes is the value of the argument to the `mpirun -np` option. In this case, *N* is an integer from 0-3. After the MPI rank children are launched, the **Multiprocess Explorer** displays them as "Stopped while sleeping within the read system call" and focuses the Main View on the MPI rank 0 process.

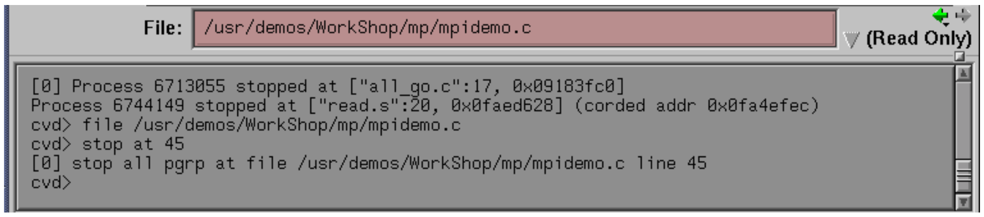
5. At this point, you are now ready to set a breakpoint in `mpidemo.c`.

   ```
   cvd> file mpidemo.c
   ```

   This brings the *mpidemo.c* file into the Source View pane of the MainView window.

   Issue the following command to set a breakpoint at line 45:

   ```
   cvd> stop at 45
   ```



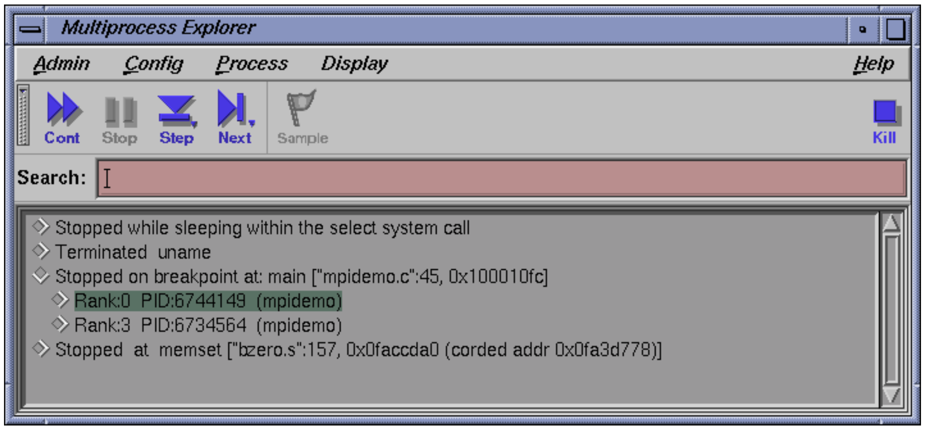
```
[0] Process 6713055 stopped at ["all_go.c":17, 0x09183fc0]
Process 6744149 stopped at ["read.s":20, 0x0faed628] (corded addr 0x0fa4efec)
cvd> file /usr/demos/WorkShop/mp/mpidemo.c
cvd> stop at 45
[0] stop all pgrp at file /usr/demos/WorkShop/mp/mpidemo.c line 45
cvd>
```

Figure 10-10 Set MPI breakpoint

6. Continue running the demo with the `Continue` command (**all** must be selected), or use the **Multiprocess Explorer continue** button::

   ```
   cvd> continue
   ```

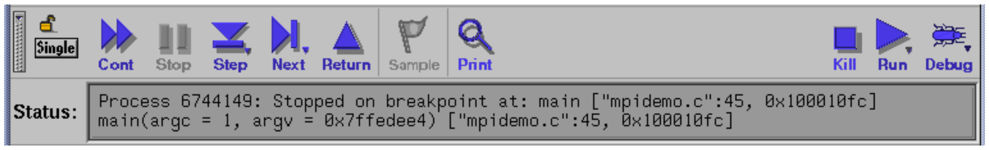
   This continues all the MPI processes and the MPI rank chilren until one (or more) of the rank children hits the breakpoint at line 45 of `mpidemo.c`. The following screen shows the **Display by Status** mode with the node entry for the processes "Stopped on breakpoint at: main ["mpidemo.c":45,0x100010fc]" opened.



```
◇ Stopped while sleeping within the select system call
◇ Terminated  uname
◇ Stopped on breakpoint at: main ["mpidemo.c":45, 0x100010fc]
   ◇ Rank:0  PID:6744149  (mpidemo)
   ◇ Rank:3  PID:6734564  (mpidemo)
◇ Stopped  at  memset ["bzero.s":157, 0x0faccda0 (corded addr 0x0fa3d778)]
```

Figure 10-11 Multiprocess Explorer: **Display by Status**

7. The **Multiprocess Explorer** focuses the Main View window on the first MPI rank process that reaches the breakpoint.



```
Process 6744149: Stopped on breakpoint at: main ["mpidemo.c":45, 0x100010fc]
main(argc = 1, argv = 0x7ffedee4) ["mpidemo.c":45, 0x100010fc]
```

Figure 10-12 MPI rank process status

Note: Depending on system load and other OS issues, some MPI rank children may not reach the breakpoint. If you need all rank children (or a specific rank child) to be at the breakpoint, you will need to put Main View into single mode and then use the **Multiprocess Explorer** "Change MainView focus to this entry" methods to select a rank child process not yet at the breakpoint and issue a **continue** (single) from Main View.

---

8. You can use the **Multiprocess Explorer continue** button to continue program execution to termination (depending on where the rank children processes are stopped, more than one **continue** may be needed) or use the **Multiprocess Explorer kill** button to terminate this example.

Debugging an OpenMP Application

Improvements have been made to OpenMP debugging for the WorkShop 2.9.2 release. The **Array Visualizer**, **Data Browser**, **Variable Browser**, and command line views now display OpenMP shared and private variables accurately, across all storage types and language nuances, to truly reflect parallel processing. Correct evaluation and display of shared and private entities in both `$omp parallel` and `$omp do` regions, across pertinent WorkShop views, for both C/C++ (especially in the use of stack variables) and FORTRAN 77 and Fortran90, have been addressed.

Stability improvements have been made in both the client and server portions of WorkShop `cvd` and the user will see fewer error and outright abort conditions while doing OpenMP debugging. Internal compiler errors, WorkShop view aborts, and server internal errors have been addressed extensively.

The following debug examples (one each for C and Fortran), illustrate the improvements that have been made. Use of these scenarios with earlier versions of WorkShop yields different and unsatisfactory results.

C/C++ OpenMP Debug Example

The following sample program is used with this debug example:

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void init(int *tablo, int *sum, int *indi) {
  bzero(tablo, sizeof(int) * 10);
  *sum = 0;
  *indi = 0;
}

void test_06(int ncpus) {

  int tablo[10];
  int sum = 0;
  int indi = 0;
  int mycpu;
  init(tablo, &sum, &indi);

#pragma omp parallel private(indi, mycpu) shared(sum, tablo, ncpus)
 {
  mycpu = mp_my_threadnum();
  for (indi = 0; indi < 10; indi++) {
    tablo[indi] = tablo[indi] + indi;
    sum += indi*mycpu + indi;
  }
 }
}

static void fun(int ncpus) {
  test_06(ncpus);
}

int main(void) {
  char *mp_set_numthreads_env;
  int mp_set_numthreads;

  if ((mp_set_numthreads_env = getenv("MP_SET_NUMTHREADS")) == NULL) {
    fprintf(stderr, "MP_SET_NUMTHREADS not defined\n");
    return 1;
  }

  mp_set_numthreads = atoi(mp_set_numthreads_env);
  fun(mp_set_numthreads);
  return 0;
}
```

Prior to starting the debugging session, set the number of CPUs to run:

```
 % setenv MP_SET_NUMTHREADS 4
```

1. Compile the program and use the debugger on the resulting `test` file:

   ```
   % cc -g -mp -o test test.c
   % cvd ./test
   ```

2. At the `cvd` prompt in the **Main View** window, enter the following:

   ```
   cvd> stop at 26
   ```

This sets a breakpoint in the program in the `pragma omp parallel` region.

3. Click the **run** button in the Debugger **Main View** window to run the program with the breakpoint.

4. In the Debugger **Main View** window, click on the **lock** icon on the left side of the screen. Also toggle the button below **lock** to **single**. When the lock button is set to **lock**, it ensures that the OpenMP thread that is currently in focus remains in focus. Toggling the button below the lock to **Single** ensures that the Debugger commands (`cont`, `next`, etc.) will only pertain to the focused **Single** OpenMP thread.

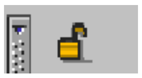


Figure 10-13 Main View Unlock icon

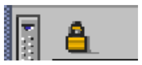


Figure 10-14 Main View Lock icon

5. At the `cvd` command prompt at the bottom of the **Main View** window, enter the following commands:

```
cvd> print mycpu
```

`mycpu` is a private variable, unique to a master/slave process. OmpThread0 (master) should be first to reach the breakpoint.

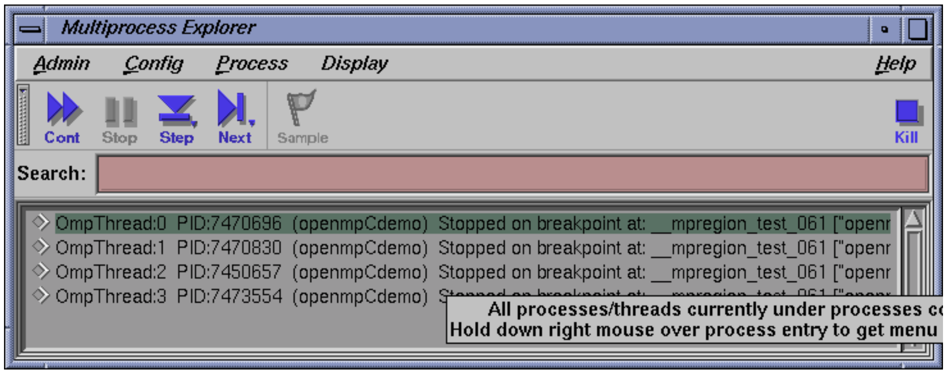
```
cvd> next 10 times
```

Again, via the **lock** button, select **single process** and **lock**.

```
cvd> print sum
```

This is a shared variable that sums across all CPUs; it should return 6 (or a number close to that) after the previous `10 next` command. It will not change when shifting focus (via **Multiprocess Explorer**) to another CPU.

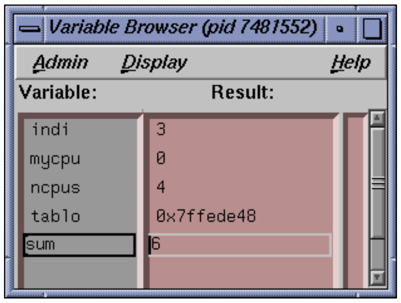
6. Select **Admin > Multiprocess Explorer** to bring up the **Multiprocess Explorer**. Note in the **Multiprocess Explorer** window that OmpThread0 is the 'master' and the other 'OmpThread' entries are the true 'slaves'. All should be stopped on the breakpoint.



```
OmpThread:0  PID:7470696  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
OmpThread:1  PID:7470830  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
OmpThread:2  PID:7450657  (openmpCdemo)  Stopped on breakpoint at: __mpregion_test_061 ["openr
OmpThread:3  PID:7473554  (openmpCdemo)  S
All processes/threads currently under processes c
Hold down right mouse over process entry to get menu
```

Figure 10-15 Multiprocess Explorer: OMP threads stopped at breakpoints

7. Select **Views > Variable Brower**. Note all 5 local variables are displayed correctly. Because you are still focused on OmpThread0 (the master) in the **MainView** window, the value of the private (for OmpThread0 only) variable `indi` should be 3 (approximately) and the value of the shared (all OmpThreads) variable `sum` should be approximately 6 after the `next 10` commands issued previously.



Variable:	Result:
indi	3
mycpu	0
ncpus	4
tablo	0x7ffede48
sum	6

Figure 10-16 Variable Browser

8. Select **Views > Data Explorer** in the **Main View** window. Click on the private variable `indi` and the shared variable `sum`. Only the private variable should change with the next step.

9. In the **Multiprocess Explorer** window, select one of the slave proceses (for example, OmpThread2) by using the left mouse to highlight the name.

   Note in the **Variable Browser** that `indi` now has a value of 0 because OmpThread2 has not been stepped. The shared variable `sum` still has a value of 6 and the private variable `mycpu` has a value of 2 to reflect the OmpThread2 now in focus in the **Main View** window.

   Note in the **Data Explorer** window that `indi`, `sum` and `mycpu` have same values as those shown in the **Variable Browser**.

10. In the `cvd` command line of the **Main View** window, enter the following commands:

    ```
    cvd> print mycpu
    ```

    The value will be 2, showing the “private” nature of `mycpu`.

    Make sure that the **single** and the **lock** are chosen on the **lock** icon. Only `OmpThread2` will advance.

    ```
    cvd> next 15 times
    cvd> print indi
    ```

    The value returned should be approximately 5. This private value is unique to `OmpThread2`.

    ```
    cvd> print sum
    ```

    The value returned should be approximately 66. This shared value is accessible to all OmpThreads.

    Note in the **Variable Browser** and the **DataExplorer** that the values for `mycpu`, `indi` and `sum` agree with those printed in the `cvd` command line with the `print` command.

11. Using the **Multiprocess Explorer**, click the right mouse button over the process entry to switch the **MainView** focus back to the master process (OmpThread0).

12. In the `cvd` command line portion of the **Main View** window, enter the following commands:

```
cvd> print mycpu
```

The value should be 0 for `OmpThread0`, which is a private variable.

```
cvd> print indi
```

This should still be 3 because OmpThread2 (not OmpThread0) was just recently stepped in the sequence of 15 steps. OmpThread0's private value of `indi` cannot change while stepping `OmpThread2`.

```
cvd> print sum
```

This is a shared variable, so it should still show changes from the next 15 times for `OmpThread2` above. Its value should still be about 66.

Note in the **Variable Browser** and **Data Explorer** that the values for `mycpu`, `indi` and `sum` agree with those printed in the command line with the `print` command.

This is the end of the C/C++ OpenMP debugging example. If this is used with WorkShop versions prior to 2.9.2, unsatisfactory results ocur.

To exit this example, select **Admin > Exit** from any Debugger window.

Fortran OpenMP Debug Example

The following sample program is used in this example:

```
program main
      implicit none
      integer n,m,mits
      double precision tol,relax,alpha

      common /idat/ n,m,mits
      common /fdat/tol,alpha,relax

      n = 3
      m = 5
      alpha = 4.4444
      relax = 3
      tol = 10
      mits = 50
      call driver ()
      stop
```

```
end

subroutine driver ( )
implicit none

integer n,m,mits,mtemp
double precision tol,relax,alpha

common /idat/ n,m,mits,mtemp
common /fdat/tol,alpha,relax

double precision u(n,m),f(n,m),dx,dy
call initialize (n,m,alpha,dx,dy,u,f)
call jacobi (n,m,dx,dy,alpha,relax,u,f,tol,mits)
return
end

subroutine initialize (n,m,alpha,dx,dy,u,f)
implicit none
integer n,m
double precision u(n,m),f(n,m),dx,dy,alpha
integer i,j, xx,yy
double precision PI
parameter (PI=3.1415926)

dx = 2.0 / (n-1)
dy = 2.0 / (m-1)
do j = 1,m
  do i = 1,n
    xx = -1.0 + dx * dble(i-1)        ! -1 < x < 1
    yy = -1.0 + dy * dble(j-1)        ! -1 < y < 1
    u(i,j) = 0.0
    f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy) -2.0*(1.0-xx*xx)-2.0
  enddo
enddo
return
end

subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)
implicit none
integer n,m,maxit
```

```
      double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
      integer i,j,k,l
      double precision error,resid,ax,ay,b
      double precision error_local, uold(n,m)

      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff
      error = 10.0 * tol
      k = 1
      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
!$omp parallel
!$omp do
         do j=1,m
            do i=1,n
               uold(i,j) = u(i,j)
            enddo
         enddo
!$omp do private(resid) reduction(+:error)
         do l = 2,m-1
            do i = 2,n-1
      resid = (ax*(uold(i-1,l)+uold(i+1,l))+b*uold(i,l)-f(i,l))/b
               u(i,l) = uold(i,l) - omega * resid
               error = error + resid*resid
            end do
         enddo
!$omp enddo nowait
!$omp end parallel
         k = k + 1
         error = sqrt(error)/dble(n*m)
      enddo                     ! End iteration loop
      print *, 'Total Number of Iterations ', k
      print *, 'Residual                   ', error
      return
      end
```

Prior to starting the debugging session, set the number of CPUs to run:

```
% setenv MP_SET_NUMTHREADS 5
```

1. Compile the program and use the debugger on the resulting `test` file:

   ```
   % f90 -g -mp -o test test.f90
   % cvd ./test
   ```

2. At the `cvd` prompt in the **Main View** window, enter the following:

   ```
   cvd> stop at 74
   ```

   This stops the program in the parallel region.

3. Click the **run** button in the **Main View** window to run the program with the breakpoint.

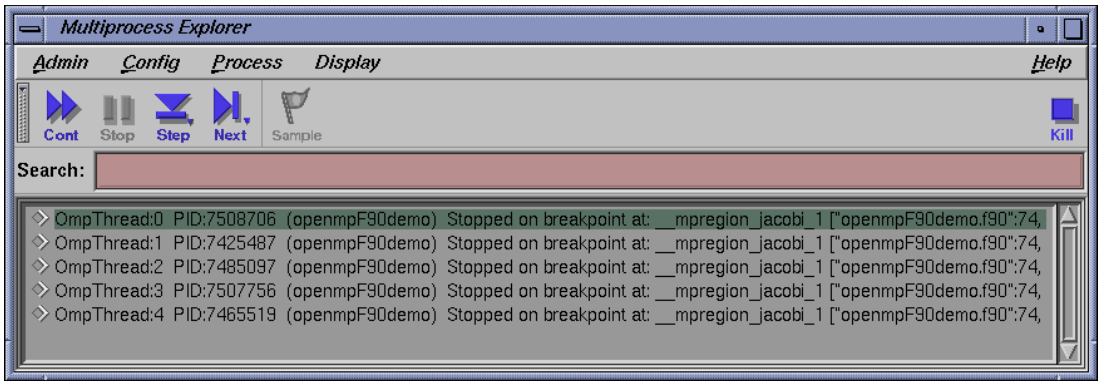
4. At the `cvd` command prompt at the bottom of the **Main View** window, enter the following commands:

   ```
   cvd> print f
   cvd> print j
   ```

   `f` is an atuomatic array with a default of shared. The bounds are dynamically set to 3x5. The value of `j` should be 1 because no stepping has been done yet.

5. Select **Views > Array Browser**. Enter `f` for the array and you will see that the values displayed are identical to those shown via the command line commands used previously.

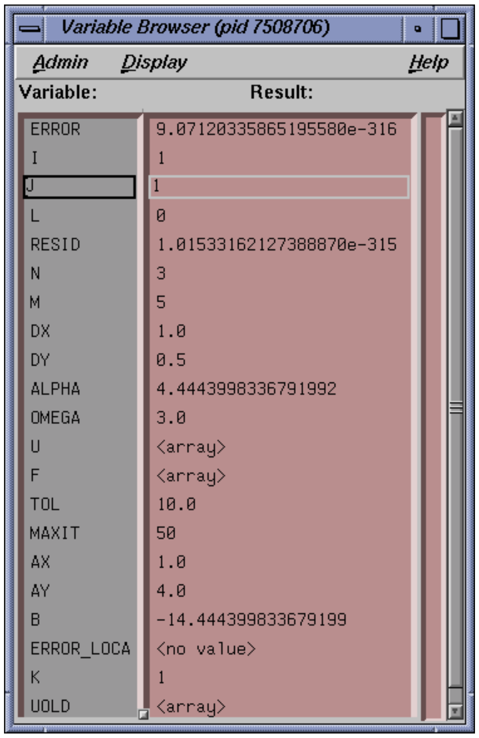
6. Select **Admin > Multiprocess Explorer**. Notice that `OmpThread0` is the 'master' and the other 'OmpThread' entries are the true 'slaves'. All threads should be stopped on the breakpoint. There should be 5 of these corresponding to the `MP_SET_NUMTHREADS 5` command issued previously. Each iteration of the outer `j` loop has been allocated to a separate thread. The **Variable Browser** (used in the next step) confirms this.



```
OmpThread:0  PID:7508706  (openmpF90demo)  Stopped on breakpoint at: __mpregion_jacobi_1 ["openmpF90demo.f90":74,
OmpThread:1  PID:7425487  (openmpF90demo)  Stopped on breakpoint at: __mpregion_jacobi_1 ["openmpF90demo.f90":74,
OmpThread:2  PID:7485097  (openmpF90demo)  Stopped on breakpoint at: __mpregion_jacobi_1 ["openmpF90demo.f90":74,
OmpThread:3  PID:7507756  (openmpF90demo)  Stopped on breakpoint at: __mpregion_jacobi_1 ["openmpF90demo.f90":74,
OmpThread:4  PID:7465519  (openmpF90demo)  Stopped on breakpoint at: __mpregion_jacobi_1 ["openmpF90demo.f90":74,
```

Figure 10-17 Multiprocess Explorer: OmpThreads stopped at breakpoint

7. Select **Views > Variable Browser**. Notice that the value of `j` should be 1 because the OmpThread0 (master) thread gets the first iteration of the `j` loop.



Variable:	Result:
ERROR	9.07120335865195580e-316
I	1
J	1
L	0
RESID	1.01533162127388870e-315
N	3
M	5
DX	1.0
DY	0.5
ALPHA	4.4443998336791992
OMEGA	3.0
U	<array>
F	<array>
TOL	10.0
MAXIT	50
AX	1.0
AY	4.0
B	-14.444399833679199
ERROR_LOCA	<no value>
K	1
UOLD	<array>

Figure 10-18 Variable Browser display

8. Select **Views > Data Explorer** and click on the variable `j`. Notice that the value agrees with that from both the **Variable Browser** and from the command line `print` command.

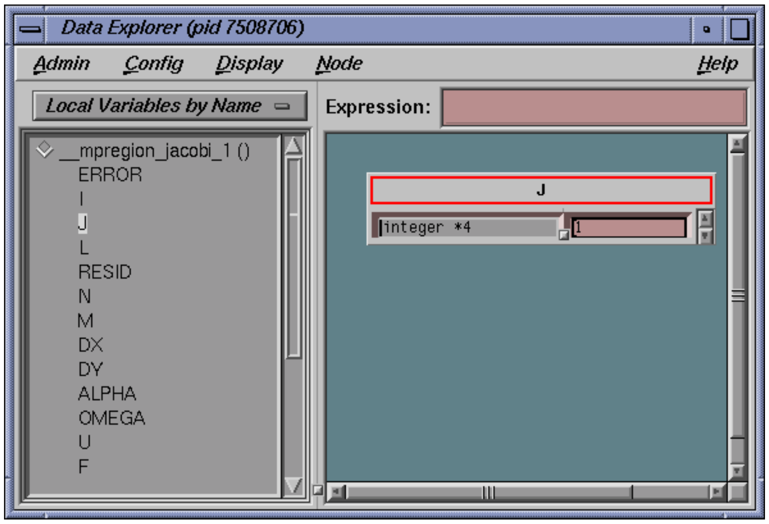


Figure 10-19 Data Explorer

Note that the lists of variables for the **Data Explorer** and the **Variable Browser** windows agree in length, name, etc.

9. In the **Main View** window, make sure that the **single** and the **lock** are chosen on the **lock** icon to ensure that a single OpenMP thread will be affected. In the `cvd` command line of the **Main View** window, enter the following commands:

   ```
   cvd> next 4 times
   ```

   ```
   cvd> print i
   ```

   The value returned should be 3 after the `next 4` commands issued previously.

   ```
   cvd> print uold
   ```

   This is an automatic, stack-based array; the default is shared. The bounds are dynamically set to 3x5. Note that the previous `next` commands have initialized a portion of `uold` to zeroes from another automatic array `u`.

10. Using the **Multiprocess Explorer**, select one of the slave proceses (for example, OmpThread2) using the left mouse button to highlight the thread. The value of `j` (should be 3 for OmpThread2) shown in the **Variable Browser** should correspond

to the iteration of the outer `j` loop assigned to this slave. The value of `i` should be 1 because this slave (OmpThread2) has not been stepped yet.

11. In the `cvd` command line portion of the **Main View** window, enter the following commands:

    ```
    cvd> print B
    ```

    This is an OpenMP shared variable whose value does not change when switching threads. Note its value.

    Make sure that the **single** and the **lock** are chosen on the **lock** icon to ensure that a single OpenMP thread will be affected.

    ```
    cvd >next 2 times
    cvd >print i
    ```

    The value returned should be 2 after the 2 `next` commands previously. This value of `i` is private, or unique, to the slave that is in focus (that is, `OmpThread2`).

12. Using the **Multiprocess Explorer**, click the right mouse button over the process entry to switch the **MainView** focus back to the master process (`OmpThread0`).

    Note in the **Variable Browser** and **Data Explorer** that the values for `i` and `j`, private to each master/slave, now change to reflect the new OpenMP thread chosen (OmpThread0). and `sum` agree with those printed in the command line with the `print` command. Variables that are shared (for example, `B`) should not change value.

This concludes the Fortran example. To exit this example, select **Admin > Exit** from any Debugger window.

OpenMP Debugging Tips

This section contains some pointers on using the Debugger on code that contains OpenMP constructs.

Setting Breakpoints in a Parallel Region

It is best to select **Group Trap Default** for your Traps Preference. That way, if you delete a breakpoint, the deletion will occur in the master and all the slaves, not just the current master or slave. This saves having to delete the breakpoint for every slave.

OMP_DYNAMIC daemon process

The daemon process used by OpenMP to handle creation and annihilation of threads in support of `OMP_DYNAMIC` is problematic for the WorkShop Debugger. This daemon is present by default and unless you are using the `schedule dynamic` feature of the OpenMP specification, it is unnecessary. The daemon does not allow clean termination of a debugging session and will end with itself in a `SIGTERM` error and the master ’hung’.

No known actual functionality is lost with this circumstance but it can be unnerving to the user. So, unless you are using `schedule dynamic` in your OpenMP program, issue a `setenv OMP_DYNAMIC false` command at the beginning of your debugging session. This allows clean termination of the master and all slaves and eliminates the `SIGTERM` error for the daemon process itself.

Consecutive and/or Nested OpenMP regions

Consider the following OpenMP C code snippet:

```
#pragma omp parallel .....
  {
     <1st section code - parallel>
  }
  ...
  ... intervening code
  ...
#pragma omp for
  for ( .... ) {
    <2nd section code - parallel??
   }
```

If a breakpoint is set in the first section of parallel code and another is set in the second section of parallel code, then only the master thread will reach the second breakpoint upon `continue` and deletion of all occurrences of the first breakpoint. All slaves will be put to ’sleep’ after doing their work in the first section of parallel code. This occurs because the notion of ’dynamic extent of an enclosing parallel region’ applies (see section 2.8 “Directive Nesting” in the OpenMP C/C++ 2.0 standard or section 2.9 “Directive Nesting” in the OpenMP Fortran 2.0 standard). There is a clause in these sections which states:

Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region. When executed outside a user-specified parallel region, the directive is executed by a team composed of only the master thread.

Thus the second section of code above will only be run by the master. Note that a similar test case for Fortran could be built using `!$omp do` (and **not** `!$ omp parallel do`). The second pragma as written is legal, but will not in fact be executed in parallel as it is **not** within the 'dynamic extent of an enclosing parallel region'; that is, it is **not** already in the middle of a parallel region that started somewhere up the call chain.

What probably occurred here is an error, but it is legal. The user probably assumed or wanted parallel execution (that is, . master **and** slaves) in the second section of code. The trouble here is that the second pragma should read as follows to ensure parallel execution:

```
 #pragma omp parallel for
```

Note the addition of `parallel`. This allows both master and slaves to reach the breakpoint in the second section of code.

Unexpected Stops in Routine nsproc

On occasion the user might find the master or slave stopped (use **MpView** to see this) in internal routine `nsproc`. This is harmless and can be remedied immediately with a **Continue All** in the **MpView**. This occurs because, as noted in "Consecutive and/or Nested OpenMP regions", the debugger causes slight timing variations that would not otherwise exist. The debugger also has internal breakpoints in the routine `nsproc` because it is this routine that creates new OpenMP threads, and important information needs to be captured by the debugger at the point of OpenMP thread creation.

Creation of OpenMP Slave Threads

Consider the following OpenMP code scenario:

There exists an OpenMP region (via `#pragma omp parallel`) in a routine called `foobar`. The main routine calls `foobar()`. Assume `OMP_NUM_THREADS` is set to 4 or so. Set a breakpoint at the `CALL SITE` for `foobar` and then run to the breakpoint.

At the breakpoint, do a **stepOver** of the call to `foobar()` Using **MultiProcess View**, observe that the slave threads are asleep and the master has actually done the **stepOver**'. Reasonably, the slaves should be gone because all parallel work is within `foobar` and that work is done.

However, that's not the case. Process creation/destruction is a comparatively expensive task, and thus slaves are not created/destroyed at each parallel region. They are created once, the first time they are needed, and then kept. Setting the environment variable `MP_CREATE` causes the slave processes to be created at startup, to exist prior to the call to `foobar()`, and to continue existing after the call.

OpenMP Slave Thread Call Ctacks

The slaves do not have a "well-formed" stack trace. They call a <nested> subroutine representing the parallel region, without ever calling the enclosing parent routine.

Chapter 11

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 `$LD_LIBRARY_PATH`. See "Launching the X/Motif Analyzer", page 172 for more information regarding the correct `$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", page 172.
- "Launching the X/Motif Analyzer", page 172.
- "Navigating the Widget Structure", page 173.
- "Examining Widgets", page 176.
- "Setting Callback Breakpoints", page 178.
- "Using Additional Features of the Analyzer", page 180.
- "Ending the Session", page 184.

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 updates with the command that `cvd` is executing.

4. Click **Run** in the **Bounce** window. You 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 174).

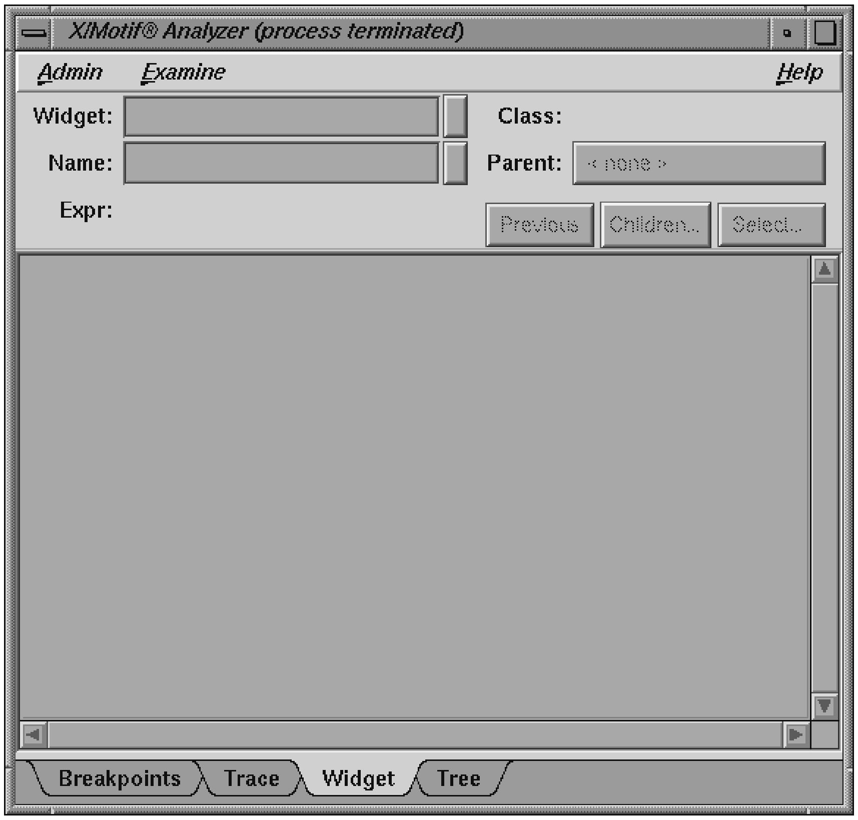


Figure 11-1 First View of the **X/Motif Analyzer** (**Widget** Examiner)

1. Widen the X/Motif Analyzer window shown in Figure 11-1, page 174. This makes it easier to understand what you are 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 do 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 175).

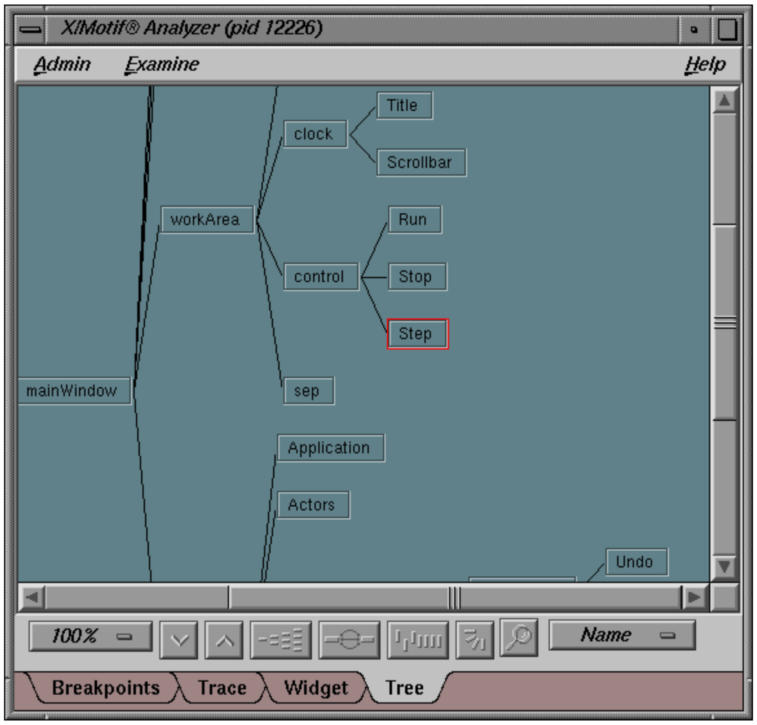


Figure 11-2 Widget Hierarchy Displayed by the Tree Examiner

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.

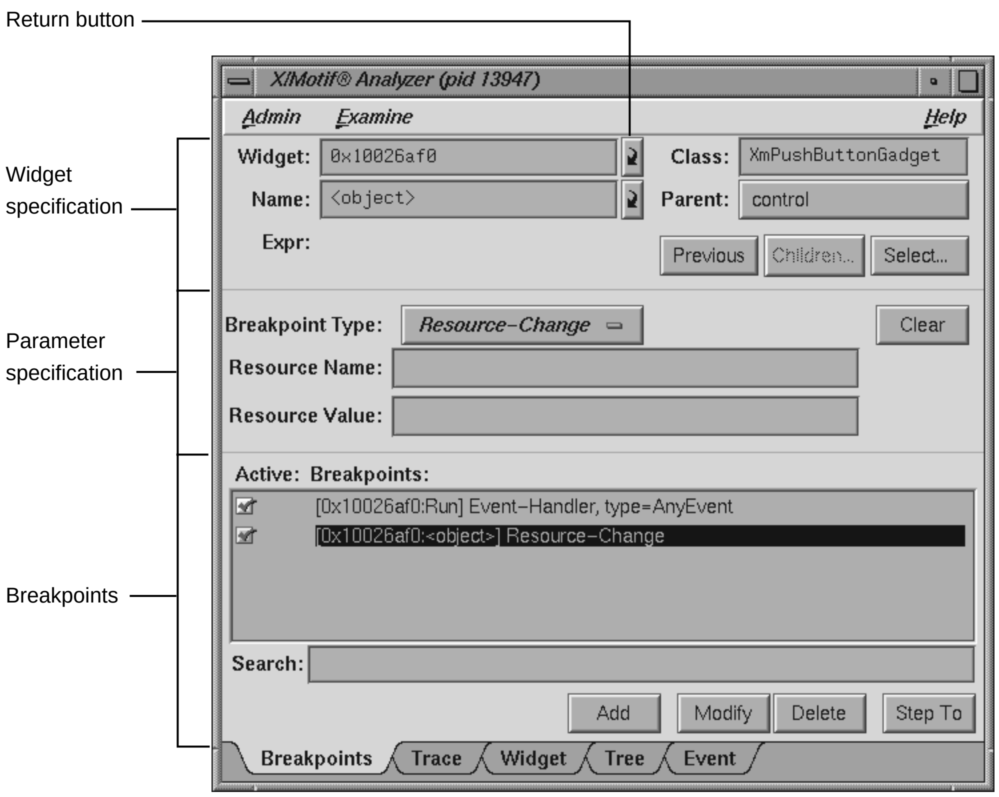
3. Enter **`stop in Clock::timeout`** at the `cvd` command line in the Main View window.

   After you press **Enter**, the red ball stops 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 177.



Widget: 0x10026af0
Name: <object>
Class: XmPushButtonGadget
Parent: control

```
[0x10026af0:Run] Event-Handler, type=AnyEvent
[0x10026af0:<object>] Resource-Change
```

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 is deleted; but the source pane still displays the arrow.

8. Click on the **Continue** button in the Main View window.

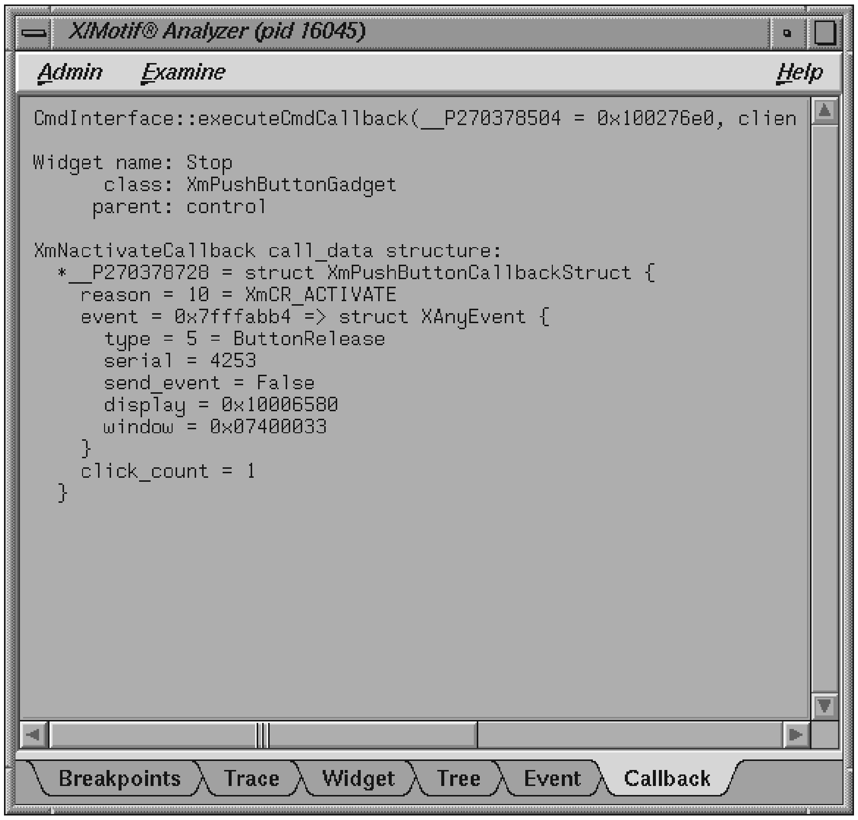
9. Click on the **Stop** button in the `Bounce` window.

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.
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 11-4).



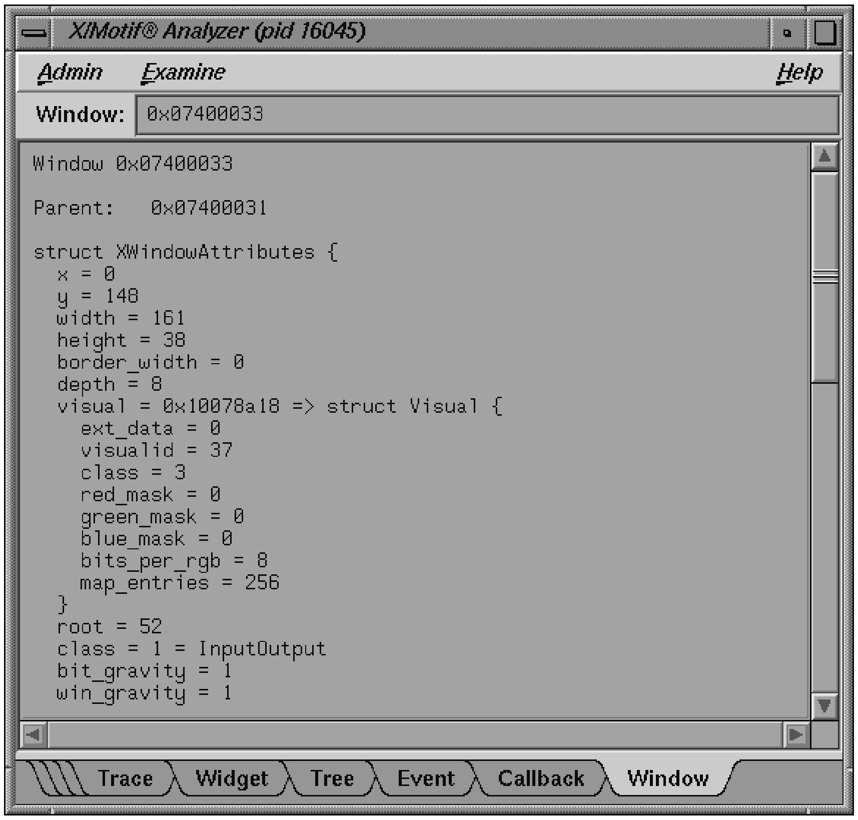
```
CmdInterface::executeCmdCallback(__P270378504 = 0x100276e0, clien

Widget name: Stop
      class: XmPushButtonGadget
     parent: control

XmNactivateCallback call_data structure:
  *__P270378728 = struct XmPushButtonCallbackStruct {
    reason = 10 = XmCR_ACTIVATE
    event = 0x7fffabb4 => struct XAnyEvent {
      type = 5 = ButtonRelease
      serial = 4253
      send_event = False
      display = 0x10006580
      window = 0x07400033
    }
    click_count = 1
  }
```

Figure 11-4 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 11-5.

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.



```
0x07400033
```

```
Window 0x07400033

Parent:   0x07400031

struct XWindowAttributes {
  x = 0
  y = 148
  width = 161
  height = 38
  border_width = 0
  depth = 8
  visual = 0x10078a18 => struct Visual {
    ext_data = 0
    visualid = 37
    class = 3
    red_mask = 0
    green_mask = 0
    blue_mask = 0
    bits_per_rgb = 8
    map_entries = 256
  }
  root = 52
  class = 1 = InputOutput
  bit_gravity = 1
  win_gravity = 1
```

Figure 11-5 Window Attributes Displayed by the Window Examiner

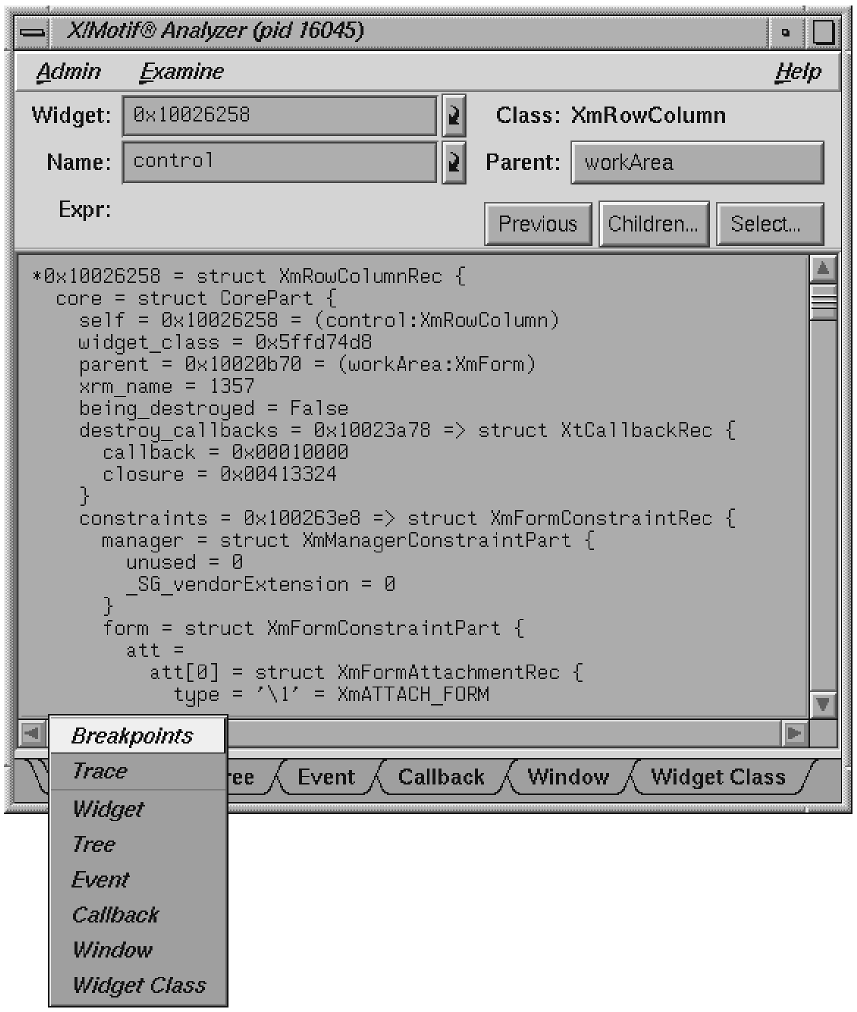
Using Additional Features of the Analyzer

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 11-6, and select the **Breakpoints** tab.



Widget: 0x10026258
Name: control
Parent: workArea

```
*0x10026258 = struct XmRowColumnRec {
  core = struct CorePart {
    self = 0x10026258 = (control:XmRowColumn)
    widget_class = 0x5ffd74d8
    parent = 0x10020b70 = (workArea:XmForm)
    xrm_name = 1357
    being_destroyed = False
    destroy_callbacks = 0x10023a78 => struct XtCallbackRec {
      callback = 0x00010000
      closure = 0x00413324
    }
    constraints = 0x100263e8 => struct XmFormConstraintRec {
      manager = struct XmManagerConstraintPart {
        unused = 0
        _SG_vendorExtension = 0
      }
      form = struct XmFormConstraintPart {
        att =
          att[0] = struct XmFormAttachmentRec {
            type = '\1' = XmATTACH_FORM
```

Figure 11-6 Selecting the Breakpoints Tab from the Overflow Area

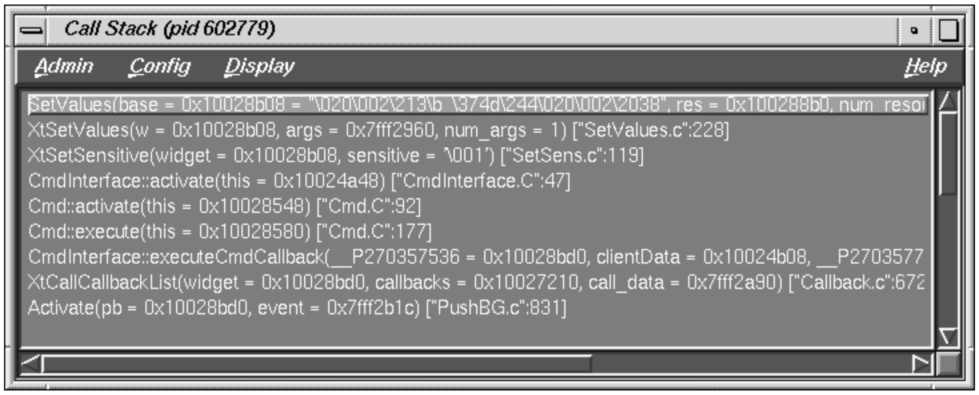
9. Click on the word **Callback** in the **Breakpoint Type** text field of the **Breakpoints Examiner** window to bring up a submenu, and select **Resource-Change**.
10. In the **Class** text field, enter: **Any**.
11. In the **Resource Name** text field, enter: **`sensitive`**.

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 updates 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 11-7).



```
SetValues(base = 0x10028b08 = "\020\002\213\b  \374d\244\020\002\2038", res = 0x100288b0, num_resoı
XtSetValues(w = 0x10028b08, args = 0x7fff2960, num_args = 1) ["SetValues.c":228]
XtSetSensitive(widget = 0x10028b08, sensitive = '\001') ["SetSens.c":119]
CmdInterface::activate(this = 0x10024a48) ["CmdInterface.C":47]
Cmd::activate(this = 0x10028548) ["Cmd.C":92]
Cmd::execute(this = 0x10028580) ["Cmd.C":177]
CmdInterface::executeCmdCallback(__P270357536 = 0x10028bd0, clientData = 0x10024b08, __P2703577
XtCallCallbackList(widget = 0x10028bd0, callbacks = 0x10027210, call_data = 0x7fff2a90) ["Callback.c":672
Activate(pb = 0x10028bd0, event = 0x7fff2b1c) ["PushBG.c":831]
```

Figure 11-7 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 updates 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 the *ProDev WorkShop: Debugger Reference Manual*.

Chapter 12

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. The definitions for the user-defined buttons display in the Debugger command line area.

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 the *ProDev WorkShop: Debugger Reference Manual*.
- `$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 Data Explorer.

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.

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.

`*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.

The following are the default font and scrollbar sizes for `cvd`:

```
cvarray*fontList: 6x13
cvdata*fontList: 6x13
cvstruct*fontList: 6x13
cvmachine*fontList: 6x13
cvmeter*fontList: 6x13
cvtaskview*fontList: 6x13
cvtaskview*boldLabelFont: 6x13
cvmp*fontList: 6x13
cvmain*fontList: 6x13
cvmotif*fontList: 6x13
cvstatic*fontList: 6x13
cvbuild*fontList: 6x13
cvpav*fontList: 6x13
cvperf*fontList: 6x13
cvxcov*fontList: 6x13
cvmake*fontList: 6x13
cvpathRemap*fontList: 6x13
*cvdHorizontalScrollBarSize: 25
*cvdVerticalScrollBarSize: 25
```

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 1
  ```

- For the Korn shell or Bourne shell, enter:

  ```
  $ DUMPCORE=;export DUMPCORE = 1
  ```

Other Variables

The user can set the following variables for the Debugger:

- `set $prompt`: specifies the prompt to be used in the command window.
- `set $addrfmt`: specifies the format for addresses.
- `set $dbxEval`: forces the command window to use the dbx evaluator for Fortran and C/C++.
- `set $dbxFortranEval`: forces the command window to use the older, dbx-like evaluator.
- `set $dbxCEval`: forces the cmmand window to use the dbx evaluator.
- `set $pendingtraps`: set to true, allows pending traps.
- `set $sprocmode`: set to true, toggles several settings suggested for debugging sproc programs.

To see complete syntax for these variables, issue the following command at the `cvd` command line:

```
cvd> help $variables
```

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).

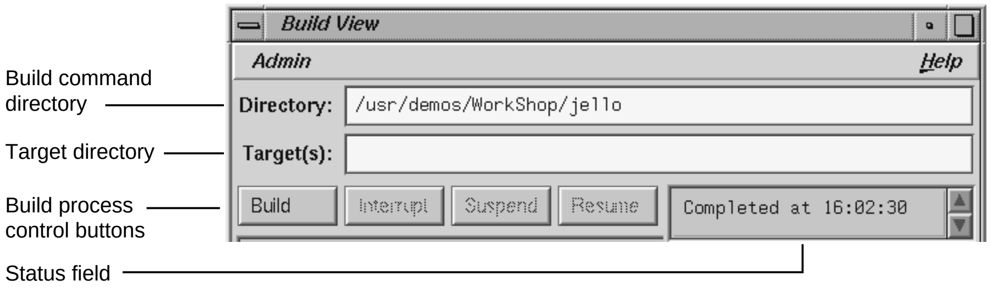
To access **Build View** from WorkShop, select **Recompile** from the **Source** menu in the Main View window in the Debugger or from the **File** menu in **Source View** (for more information on the Main View and **Source View** windows, refer to Chapter 1, "WorkShop Debugger Overview", page 1). Selecting **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 192
- "Transcript Area", page 193
- "Error List Area", page 193

Build Process Control Area

The build process control area lets you run or stop the build and view the status. See Figure A-1.



/usr/demos/WorkShop/jello

Completed at 16:02:30

Figure A-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 194, and "Build Options", page 195). The target to be built is specified in the **Target(s):** field.

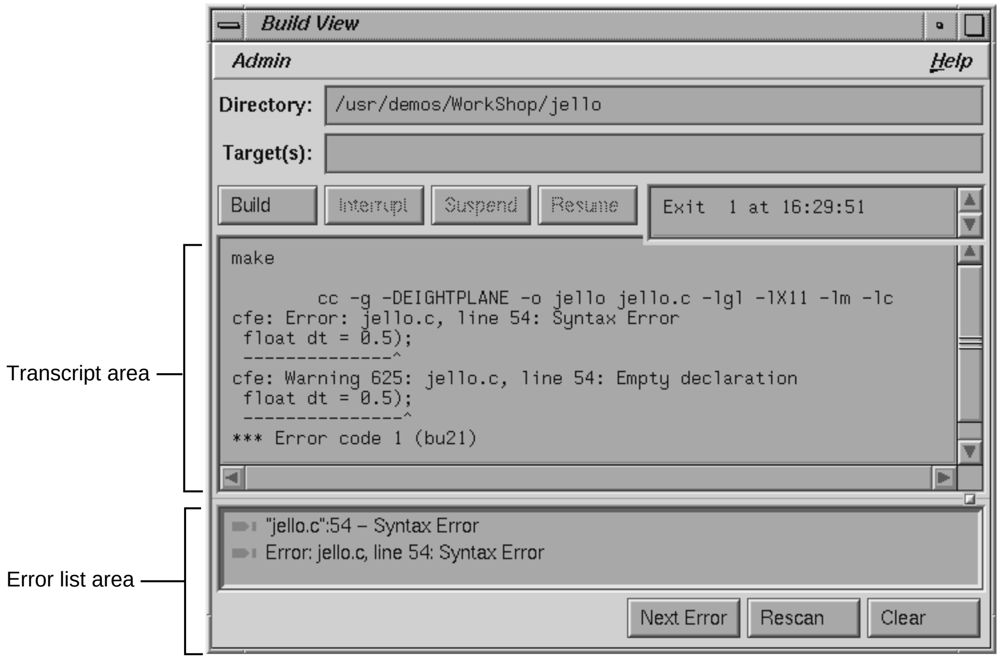
The build process control buttons let you control the build process. The following buttons are available:

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 A-2.



/usr/demos/WorkShop/jello

```
Exit  1 at 16:29:51
```

```
make

        cc -g -DEIGHTPLANE -o jello jello.c -lgl -lX11 -lm -lc
cfe: Error: jello.c, line 54: Syntax Error
 float dt = 0.5);
 --------------^
cfe: Warning 625: jello.c, line 54: Empty declaration
 float dt = 0.5);
 ---------------^
*** Error code 1 (bu21)
```

"jello.c":54 – Syntax Error
Error: jello.c, line 54: Syntax Error

Figure A-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 A-2, page 193). 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 194
- "Build Options", page 195

For information on the **Launch Tool** and **Exit** menu selections, see the *ProDev WorkShop: Debugger Reference Manual*.

Build View Preferences

The **Preferences** selection brings up the dialog box shown in Figure A-3, page 195. 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

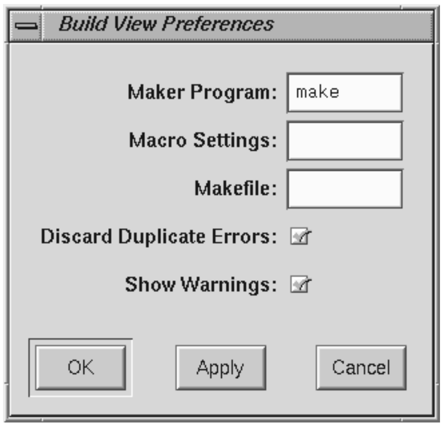
Lets you enter the name of a makefile if you do not wish to use the default.

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.



Maker Program: make
Macro Settings:
Makefile:
Discard Duplicate Errors: ✓
Show Warnings: ✓

Figure A-3 Build View Preferences Dialog

Build Options

The **Build Options Dialog** lets you add the options shown in Figure A-4, page 196, to your `make` command.

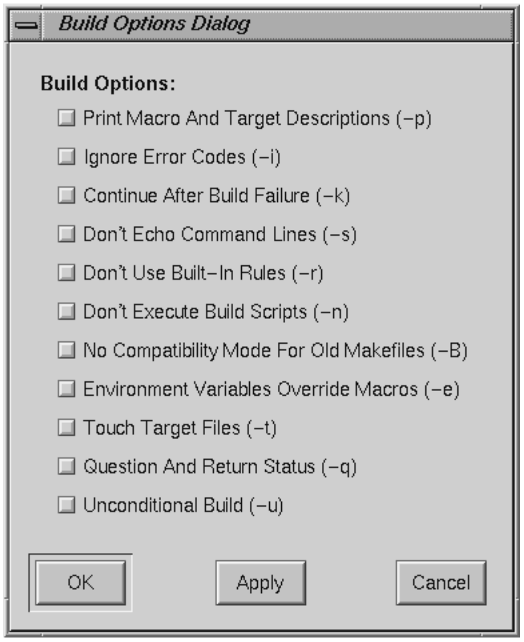

Build Options Dialog
Build Options:
Print Macro And Target Descriptions (–p)
Ignore Error Codes (–i)
Continue After Build Failure (–k)
Don't Echo Command Lines (–s)
Don't Use Built–In Rules (–r)
Don't Execute Build Scripts (–n)
No Compatibility Mode For Old Makefiles (–B)
Environment Variables Override Macros (–e)
Touch Target Files (–t)
Question And Return Status (–q)
Unconditional Build (–u)
OK
Apply
Cancel

Figure A-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.

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 A-5, page 199.

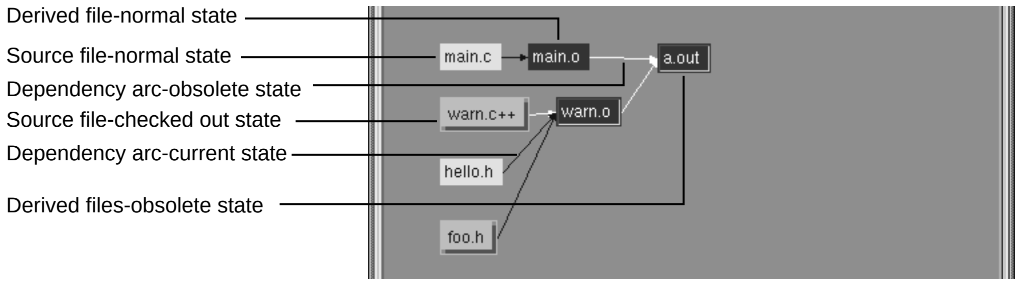

Derived file-normal state
Source file-normal state
Dependency arc-obsolete state
Source file-checked out state
Dependency arc-current state
Derived files-obsolete state
main.c
main.o
a.out
warn.c++
warn.o
hello.h
foo.h

Figure A-5 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 A-6, page 200.

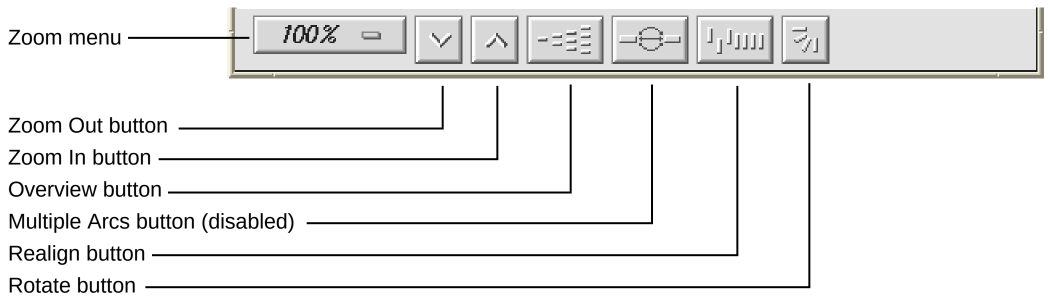


Figure A-6 Build Graph Control Area

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 A-6, page 200).

Figure A-7, page 201, 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.

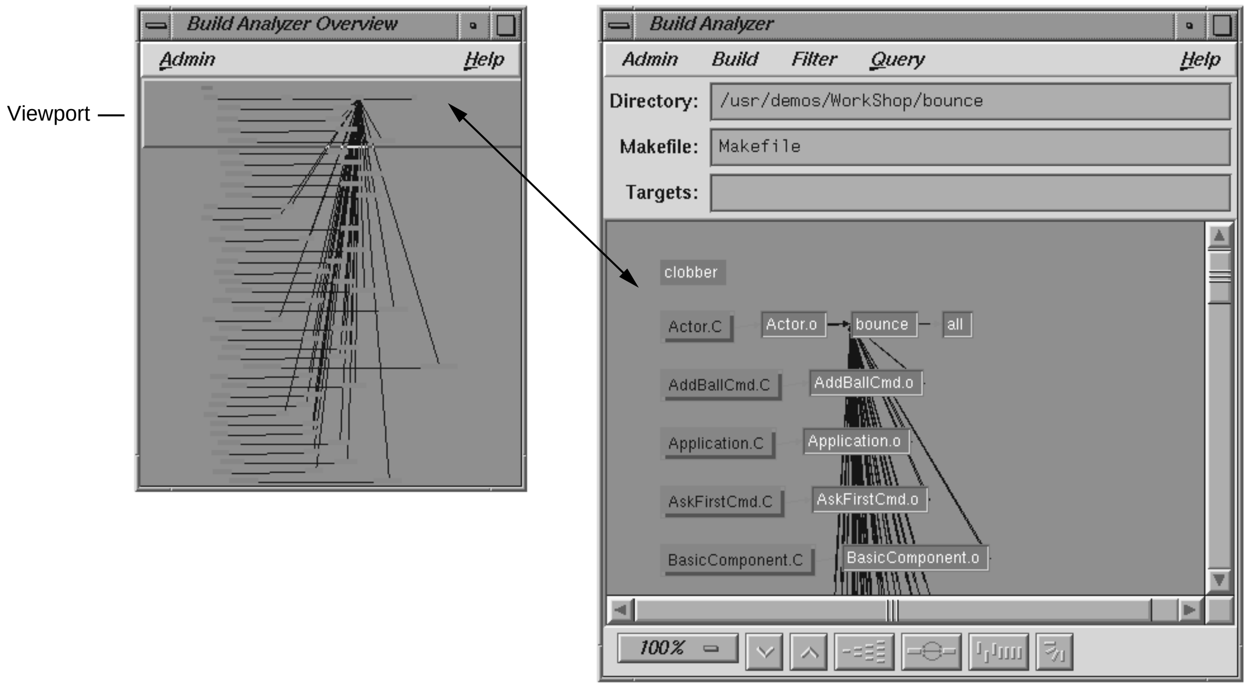


Figure A-7 Build Analyzer Overview Window with Build Analyzer Graph

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 *ProDev WorkShop: Debugger Reference Manual*.

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