Developer Magic™: ProDev WorkShop and MegaDev Overview

Document Number 007-2582-001

CONTRIBUTORS

Written and illustrated by John C. Stearns
Production by Laura Cooper
Engineering contributions by Lia Adams, Jim Ambras, Trevor Bechtel, Wes Embry, Alan Foster, Christine Hanna, David Henke, Marty Itzkowitz, Mahadevan Iyer, Lisa Kvarda, Stuart Liroff, Song Liang, Allan McNaughton, Michey Mehta, Sudhir Mohan, Ashok Mouli, Anil Pal, Andrew Palay, Tom Quiggle, Kim Rachmeler, Jack Repenning, Paul Sanville, Ravi Shankar, John Templeton, Michele Chambers Turner, Shankar Unni, Mike Yang, Jun Yu, and Doug Young.

Developer Magic™: ProDev WorkShop and MegaDev Overview
Document Number 007-2582-001

Contents

List of Figures

List of Tables

ProDev WorkShop and MegaDev Overview

Welcome to ProDev WorkShop and MegaDev, two major components in the Developer Magic™ software development environment. ProDev WorkShop contains the core software development tools; MegaDev contains advanced features for the development of C and C++ applications. These powerful, highly visual tools help you understand your program's structure and operation so that you can diagnose very difficult, traditionally time-consuming problems in a short amount of time. With them, you can develop applications for the entire Silicon Graphics® product line, from Indy™ to POWER Onyx™ workstations.

Note: In the past, the software development environment was called CASEVision™; that name has been replaced by Developer Magic. In addition to ProDev WorkShop and MegaDev, the Developer Magic environment includes ProMPF—a special module for multi-process Fortran programming—and IDO (IRIX Development Option)—the base compiler and libraries. Some of the documentation may still use the CASEVision name; those documents will be updated soon.

The ProDev WorkShop core tools provide:

- **Comprehensive control over the debugging process**—You can set simple breakpoints with the click of a mouse button or define complex conditions for your traps. ProDev WorkShop's fast data watch points with kernel support are especially adept at tracking memory corruption problems.
- **Visual debugging environment for examining data in your active program**—ProDev WorkShop provides convenient, graphical views of variables, expressions, large arrays, and data structures. If you prefer a tty-style interface, you can always dump values directly using WorkShop's Debugger command line.
- **Powerful static analysis for understanding your program**—You can view the structure of your program and relationships such as call trees, function lists, class hierarchies, and file dependencies. And you can get this information whether or not the program can be compiled.

- **The ability to collect performance and coverage information during test runs—**ProDev WorkShop's Performance Analyzer lets you see where your program spends its time and pinpoint performance bugs, including those due to memory problems. The Tester tool shows you which source lines and basic blocks are covered in your tests.
- **Convenient recompiling from within the ProDev WorkShop environment—**WorkShop's standard build tools let you view file dependencies and compiler requirements and fix compile errors conveniently.

The MegaDev tools provide:

- **Quick recompiles for simple changes—**The Fix and Continue tool lets you make simple changes without having to go through a major recompile and relinking, dramatically reducing the number of edit-compile-debug cycles.
- **Ability to analyze structures and relationships in C++ code—**The C++ Browser provides global graphical and textual views of interclass relationships, including inheritance, containment, and interactions within a set of classes.
- **Specialized debugging for IRIX IM™ applications—**MegaDev's IRIX IM Analyzer lets you solve the special problems in IRIX IM application development. You can look at object data, set breakpoints at the object or X protocol level, trace X and widget events, and tune performance.
- **Rapid application development—**The RapidApp tool lets you create graphical interfaces for C++ applications quickly and easily. RapidApp lets you build graphical interfaces by dragging and dropping interface elements (based on IRIX IM widgets and IRIS ViewKit™-style components) onto a template window.

This overview gives you a broad exposure to the ProDev WorkShop and MegaDev tools as well as pointers to the documentation for getting detailed information. The overview is organized as follows:

- "Using the ProDev WorkShop Debugger"
- "Navigating Through Code With the Static Analyzer and C++ Browser"
- "Pinpointing Performance Problems With the Performance Analyzer"

- "Determining the Thoroughness of Test Coverage With Tester"
- "Recompiling Within the ProDev WorkShop Environment With Build Manager"
- "Making Quick Changes With Fix and Continue"
- "Debugging X/Motif Programs"
- "Building Application Interfaces With RapidApp"

In addition to the ProDev WorkShop and MegaDev tools, you can separately purchase:

- **Developer Magic Pro MPF—**a visual code parallelization tool used with the Power Fortran Accelerator™ to help balance parallel loops in Fortran applications
- **Developer Magic ClearCase**™**—**a toolset for version control, configuration management, and process control for software organizations

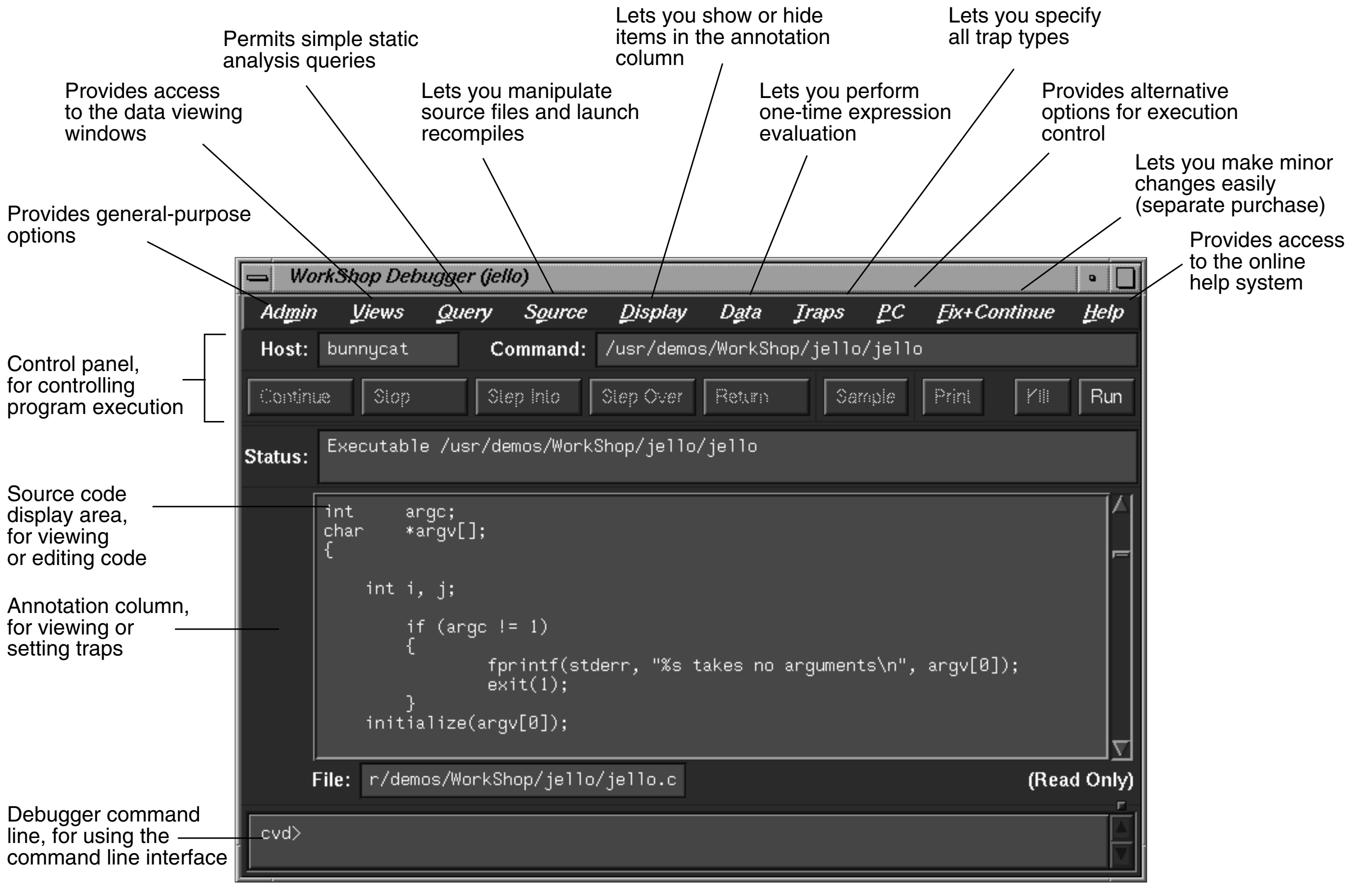
  Note: If you use ClearCase, SCCS, or RCS, you can check source files directly into or out of ProDev WorkShop and MegaDev.
- **Developer Magic Tracker—**an application builder for creating change control and change tracking systems. It can be integrated with ClearCase.

Using the ProDev WorkShop Debugger

The Debugger is a UNIX® source-level debugging tool that provides special windows (views) for displaying program data and execution state as the program executes. The Debugger lets you set various types of breakpoints and watch points where you can conveniently view data such as variables, expressions, structures, large arrays, call stacks, and machine-level values. The WorkShop Debugger goes far beyond the capabilities of *dbx*. It includes fast data watchpoints and other types of traps; graphical views for displaying local variables, source-level expressions, array variables, and data structures; and debugging at the machine level.

Debugger User Model

All WorkShop activities can be accessed from the Main View window, which is illustrated in Figure 1.


Permits simple static analysis queries
Lets you show or hide items in the annotation column
Lets you specify all trap types
Provides access to the data viewing windows
Lets you manipulate source files and launch recompiles
Lets you perform one-time expression evaluation
Provides alternative options for execution control
Lets you make minor changes easily (separate purchase)
Provides general-purpose options
Provides access to the online help system
Control panel, for controlling program execution
Source code display area, for viewing or editing code
Annotation column, for viewing or setting traps
Debugger command line, for using the command line interface
WorkShop Debugger (jello)
Admin
Views
Query
Source
Display
Data
Traps
PC
Fix+Continue
Help
Host:
Command:
Continue
Stop
Step Into
Step Over
Return
Sample
Print
Kill
Run
Status:
File:
(Read Only)

Host: bunnycat
Command: /usr/demos/WorkShop/jello/jello
Status: Executable /usr/demos/WorkShop/jello/jello

```
int     argc;
char    *argv[];
{

    int i, j;

        if (argc != 1)
        {
                fprintf(stderr, "%s takes no arguments\n", argv[0]);
                exit(1);
        }
    initialize(argv[0]);
```

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

```
cvd>
```

Figure 1 Major Areas of the Main View Window

The basic model for using the Debugger is to:

1. Invoke the Debugger by typing:

   `cvd [-pid` *pid*`] [-host` *host*`] [`*executable* `[`*corefile*`]] [&]`

The **-pid** option lets you attach the Debugger to a running process. You can use this to determine why a live process is in an infinite loop or is otherwise hung.

The argument *executable* is the name of the executable file for the process you want to run. It is optional; you can invoke the Debugger first and specify the executable later.

The *corefile* option lets you invoke the Debugger and specify a core file (with its executable) to try to determine why a program crashed.

The **-host** option lets you specify a remote host on which the target executable will be run; the Debugger runs locally. This option is useful if:

- you don't want the Debugger windows to interfere with the application you are debugging.
- you are supporting an application remotely.
- you don't want to use the Debugger on the target system for another reason.

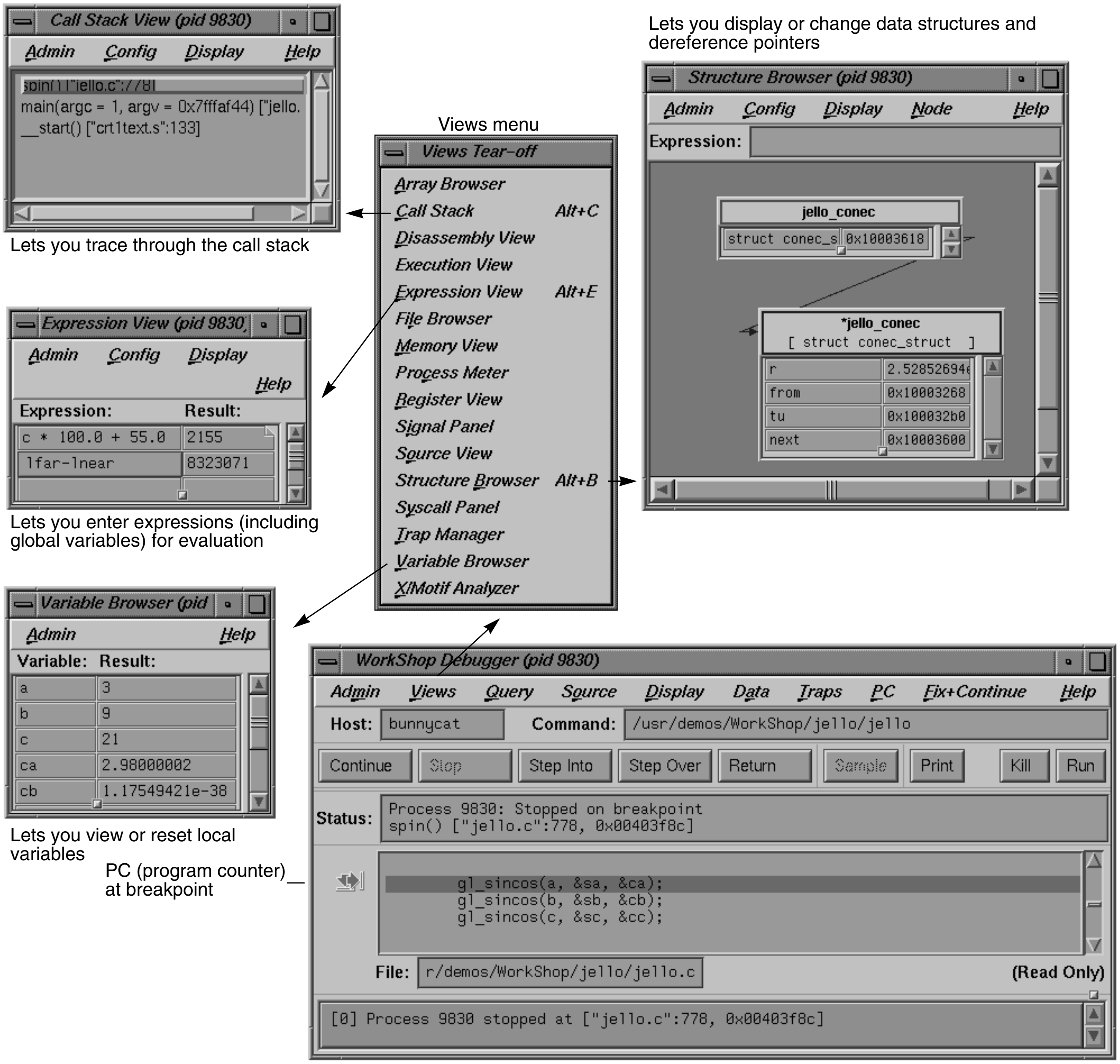
2. Set stop traps, that is, breakpoints, in the source code.

   Simple traps are set by clicking the left mouse button in the annotation column to the left of the source code display or by using the Traps menu. More complex traps, including watch points, can be set and managed from the Trap Manager, Signal Panel, and Syscall Panel, which can be accessed from the Views menu. You can also set traps by typing them at the Debugger command line in Main View. You can stop a process at any time by clicking the *Stop* button in the Main View control area.

3. Start the program by clicking the *Run* button in Main View.

4. When the process stops at a breakpoint or other stopping point of interest, you can examine the data in the Debugger view windows (accessed from the Views menu).

   You can display view windows at any time; they update automatically each time the program stops. Figure 2 shows four typical Debugger views and indicates how you access them from the Views menu.


Call Stack View (pid 9830)
Admin Config Display Help
Lets you trace through the call stack
Expression View (pid 9830)
Admin Config Display Help
Lets you enter expressions (including global variables) for evaluation
Variable Browser (pid
Admin Help
Lets you view or reset local variables
PC (program counter) at breakpoint
Views menu
Views Tear-off
Array Browser
Call Stack Alt+C
Disassembly View
Execution View
Expression View Alt+E
File Browser
Memory View
Process Meter
Register View
Signal Panel
Source View
Structure Browser Alt+B
Syscall Panel
Trap Manager
Variable Browser
X/Motif Analyzer
Lets you display or change data structures and dereference pointers
Structure Browser (pid 9830)
Admin Config Display Node Help
Expression:
jello_conec
*jello_conec
WorkShop Debugger (pid 9830)
Admin Views Query Source Display Data Traps PC Fix+Continue Help
Continue Stop Step Into Step Over Return Sample Print Kill Run
(Read Only)

```
spin() ["jello.c":778]
main(argc = 1, argv = 0x7fffaf44) ["jello.
__start() ["crt1text.s":133]
```

Expression:	Result:
c * 100.0 + 55.0	2155
lfar-lnear	8323071

Variable:	Result:
a	3
b	9
c	21
ca	2.98000002
cb	1.17549421e-38

```
struct conec_s  0x10003618
[ struct conec_struct ]
```

r	2.52852694
from	0x10003268
tu	0x100032b0
next	0x10003600

Host: bunnycat
Command: /usr/demos/WorkShop/jello/jello

Status:
```
Process 9830: Stopped on breakpoint
spin() ["jello.c":778, 0x00403f8c]
```

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

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

```
[0] Process 9830 stopped at ["jello.c":778, 0x00403f8c]
```

Figure 2 Typical Debugger Views Accessible at a Breakpoint

Figure 3 shows the Array Visualizer, a powerful view for examining data in arrays of up to 100 x 100 elements. You can look for problem areas in a 3D rendering of the array, click on the area of interest, and view the numerical values in a spreadsheet format. In Figure 3, the hue option has been set so that the values appear in a color spectrum from blue (lowest) to red (highest) with out-of-range anomalies appearing in gray. Note the high point coming out of the 3D image; it demonstrates how anomalies in large arrays stand out.

If you need to debug your program at the machine level, you can use Register View, Disassembly View, and Memory View, as shown in Figure 4. These are accessed from the Views menu in the Debugger Main View as well.

5. Use the control options in Main View to continue execution (see Figure 1).

   From any breakpoint, you have these options:

   ■ The *Continue* button runs the program until the next breakpoint.

   ■ The "Continue To" selection in the PC menu proceeds to a specified source line. Placing the cursor in a line specifies it.

   ■ The "Jump To" selection in the PC menu goes to a specified line (by the cursor), skipping over any intermediate code.

   ■ The *Step Into* button continues execution by one step or a number specified by holding down the right mouse button over the *Step Into* button and selecting the number from the dialog box. The process then continues the specified number of source lines and enters any called functions.

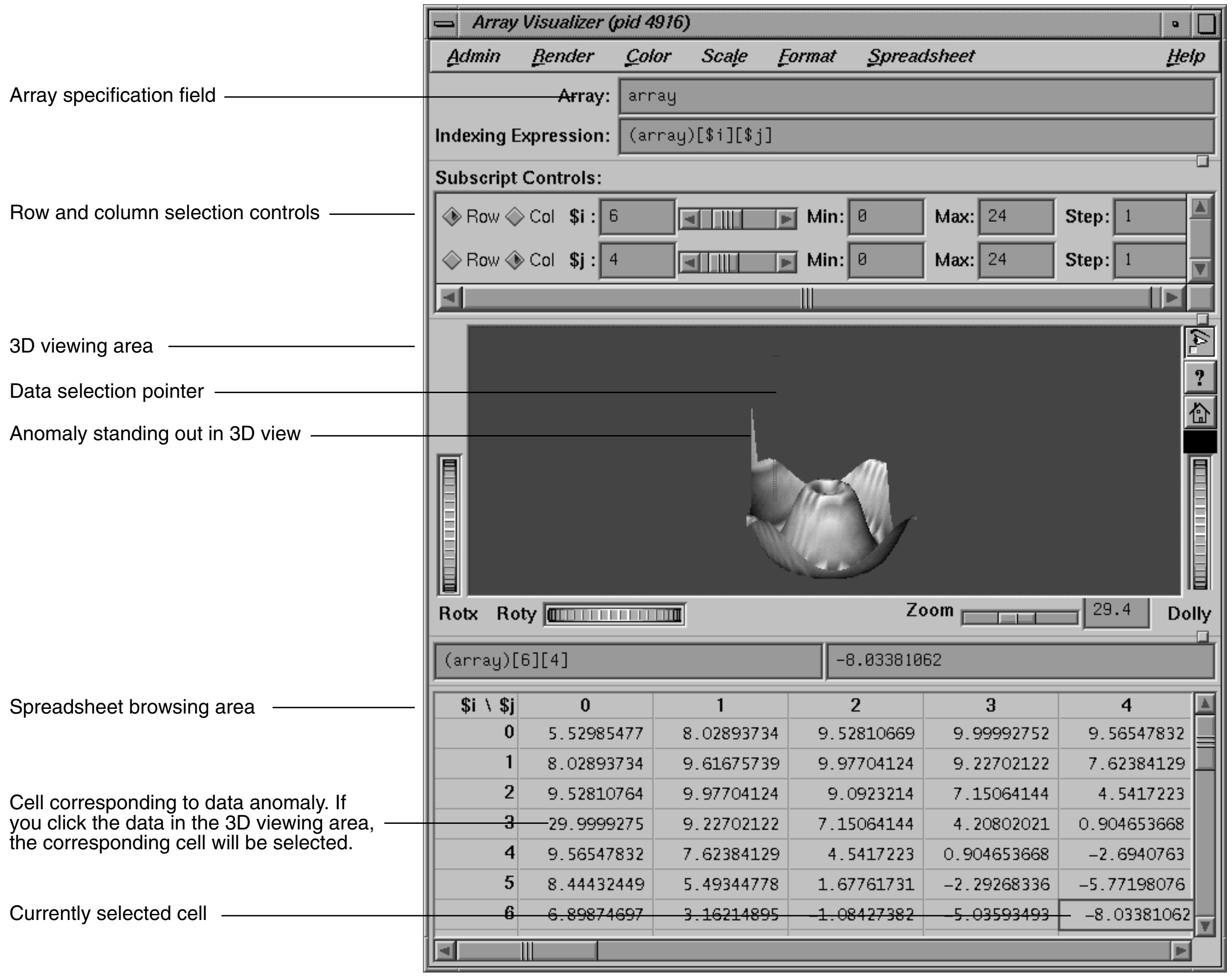
   ■ The *Step Over* button similarly proceeds a specified number of lines but executes intermediate functions without stepping into them.

   ■ The *Return* button executes the remaining instructions in a function and stops on return from that function.

6. Check out the source code that needs to be fixed.

   If you find a bug and are using an integrated source control program such as ClearCase, RCS, or SCCS, you can check out the source code from Main View (or Source View, an alternate editing window).

   Choose "Check Out" from the Versioning submenu in the Source menu.


Array Visualizer (pid 4916)
Admin
Render
Color
Scale
Format
Spreadsheet
Help
Subscript Controls:
Row
Col
Rotx
Roty
Zoom
Dolly
Array specification field
Row and column selection controls
3D viewing area
Data selection pointer
Anomaly standing out in 3D view
Spreadsheet browsing area
Cell corresponding to data anomaly. If you click the data in the 3D viewing area, the corresponding cell will be selected.
Currently selected cell

Array: array

Indexing Expression: (array)[$i][$j]

$i : 6 Min: 0 Max: 24 Step: 1

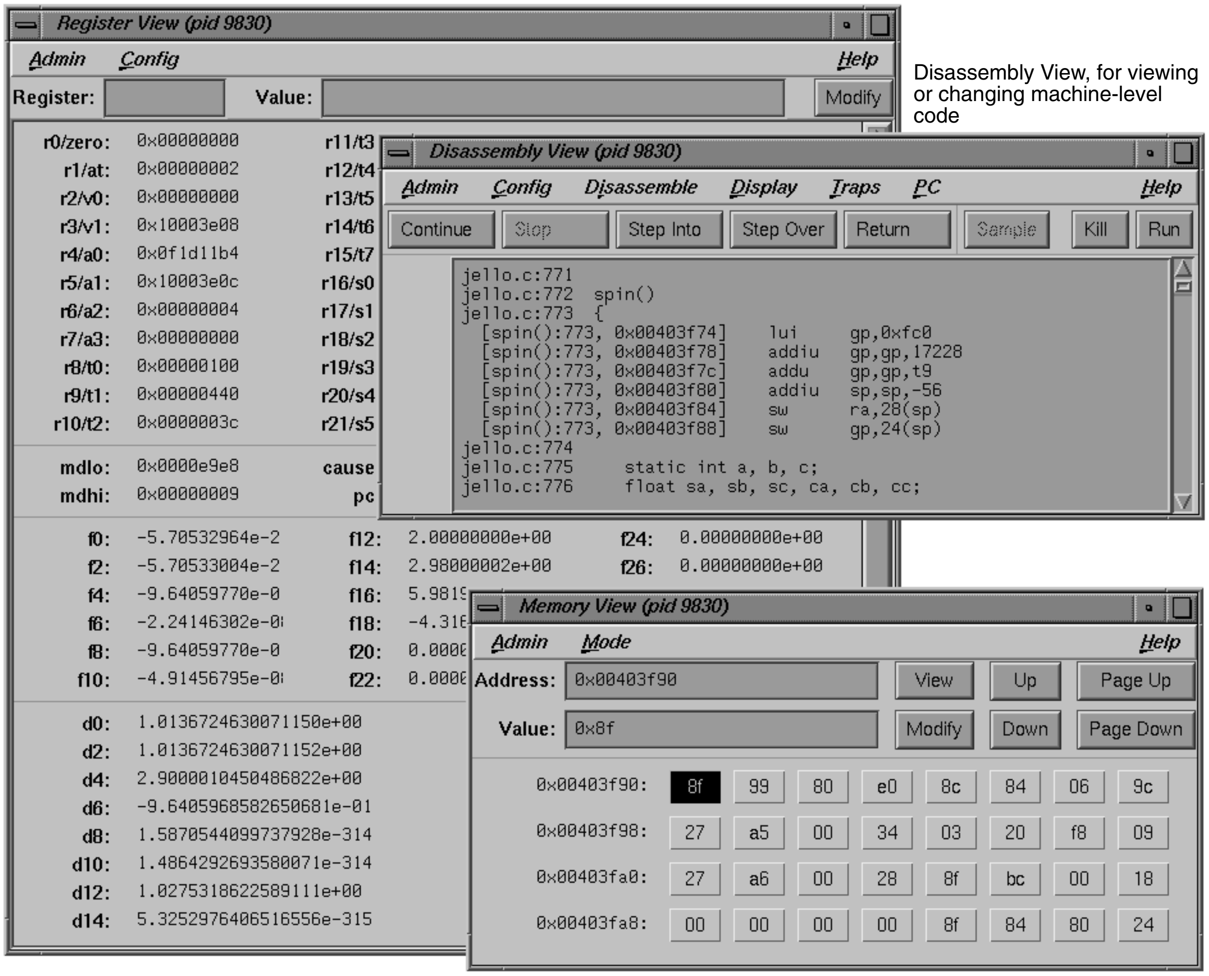
$j : 4 Min: 0 Max: 24 Step: 1

Zoom 29.4

(array)[6][4] -8.03381062

$i \ $j	0	1	2	3	4
0	5.52985477	8.02893734	9.52810669	9.99992752	9.56547832
1	8.02893734	9.61675739	9.97704124	9.22702122	7.62384129
2	9.52810764	9.97704124	9.0923214	7.15064144	4.5417223
3	29.9999275	9.22702122	7.15064144	4.20802021	0.904653668
4	9.56547832	7.62384129	4.5417223	0.904653668	-2.6940763
5	8.44432449	5.49344778	1.67761731	-2.29268336	-5.77198076
6	6.89874697	3.16214895	-1.08427382	-5.03593493	-8.03381062

Figure 3 Array Visualizer



```
 r0/zero:  0x00000000        r11/t3
   r1/at:  0x00000002        r12/t4
   r2/v0:  0x00000000        r13/t5
   r3/v1:  0x10003e08        r14/t6
   r4/a0:  0x0f1d11b4        r15/t7
   r5/a1:  0x10003e0c        r16/s0
   r6/a2:  0x00000004        r17/s1
   r7/a3:  0x00000000        r18/s2
   r8/t0:  0x00000100        r19/s3
   r9/t1:  0x00000440        r20/s4
  r10/t2:  0x0000003c        r21/s5

    mdlo:  0x0000e9e8         cause
    mdhi:  0x00000009            pc

      f0:  -5.70532964e-2     f12:  2.00000000e+00     f24:  0.00000000e+00
      f2:  -5.70533004e-2     f14:  2.98000002e+00     f26:  0.00000000e+00
      f4:  -9.64059770e-0     f16:  5.9819
      f6:  -2.24146302e-0     f18:  -4.316
      f8:  -9.64059770e-0     f20:  0.0000
     f10:  -4.91456795e-0     f22:  0.0000

      d0:  1.0136724630071150e+00
      d2:  1.0136724630071152e+00
      d4:  2.9000010450486822e+00
      d6:  -9.6405968582650681e-01
      d8:  1.5870544099737928e-314
     d10:  1.4864292693580071e-314
     d12:  1.0275318622589111e+00
     d14:  5.3252976406516556e-315
```

```
jello.c:771
jello.c:772  spin()
jello.c:773  {
  [spin():773, 0x00403f74]    lui     gp,0xfc0
  [spin():773, 0x00403f78]    addiu   gp,gp,17228
  [spin():773, 0x00403f7c]    addu    gp,gp,t9
  [spin():773, 0x00403f80]    addiu   sp,sp,-56
  [spin():773, 0x00403f84]    sw      ra,28(sp)
  [spin():773, 0x00403f88]    sw      gp,24(sp)
jello.c:774
jello.c:775     static int a, b, c;
jello.c:776     float sa, sb, sc, ca, cb, cc;
```

```
Address: 0x00403f90
  Value: 0x8f

0x00403f90:  8f  99  80  e0  8c  84  06  9c
0x00403f98:  27  a5  00  34  03  20  f8  09
0x00403fa0:  27  a6  00  28  8f  bc  00  18
0x00403fa8:  00  00  00  00  8f  84  80  24
```

Figure 4 Machine-Level Debugger Views

7. Fix any problems in your code using the source code display area in Main View, Source View, or the editor of your choice.

   Both Main View and Source View let you do simple editing and annotate the code with trap indicators. Source View also lets you display test data from the Performance Analyzer and Tester in the annotation column. If you prefer to view source code in a text editor other than Source View, add the line

   ```
   *editorCommand: editor
   ```

   to your *.Xdefaults* file, where *editor* is the command for the editor you wish to use.

8. Recompile using Build Manager.

   Build Manager has two windows: Build View and Build Analyzer. Build View lets you compile, view compile error lists, and access the offending code in Source View or an editor of your choice. Build Analyzer lets you view build dependencies and recompilation requirements, and access source files. Build View uses the UNIX *make* facility as its default build software. Although Build Analyzer determines dependencies using *make*, you can substitute the build software of your choice, any *make* that runs on Silicon Graphics platforms.

Where to Find Debugger Information

To find out more about the Debugger, refer to Table 1.

Table 1 Where to Find Debugger Information in the *CASEVision/WorkShop User's Guide*

Topic	**See ...**
General Debugger information	Chapter 1, "Getting Started with the WorkShop Debugger"
Debugger tutorial	Chapter 2, "A Short Debugger Tutorial"
Debugger interaction with source files	Chapter 3, "Debugger: Managing Source Files"
Managing windows while performing multiple tasks	Chapter 4, "Debugger: CASEVision Project Session Management"
Comprehensive trap information	Chapter 5, "Debugger: Setting Traps in WorkShop"

Table 1 **(continued)** Where to Find Debugger Information in the *CASEVision/WorkShop User's Guide*

Topic	See ...
Controlling execution in a process (stepping, jumping, etc.)	Chapter 6, "Debugger: Controlling Process Execution"
Examining Debugger data in general at the source level	Chapter 7, "Examining Debugger Data"
Tracing through the call stack	"Tracing through Call Stack View" on page 98
Entering expressions to be evaluated at stopping points	"Evaluating Expressions" on page 101
Viewing or changing the values of variables	"Examining Variable Values" on page 109
Examining data in arrays using the 3D or spreadsheet format	"Examining Array Variables" on page 111
Determining the data structures of variables	"Looking at Data Structures" on page 126
Using the Debugger command line	"Examining Data at the Command Line" on page 137
Examining debugger data at the machine level	Chapter 8, "Machine-level Debugging"
Using the debugger to trap memory allocation problems	Chapter 9, "Debugger: Detecting Heap Corruption"
Debugging multiprocess programs	Chapter 10, "Multiple Process Debugging"

Navigating Through Code With the Static Analyzer and C++ Browser

The ProDev WorkShop Static Analyzer is a source code analysis and navigation tool for analyzing source code written in C, C++, or Fortran. (The C++ Browser has additional features for C++ and is described in "C++ Browser User Model" on page 16.) The Static Analyzer shows you the code's structure (graphically or in text format) including information on function calls, definitions of variables, file dependencies, macro locations, class hierarchies, file dependencies, and many other structural details for understanding your code. In addition, you can make specific queries, such as showing everywhere a function is used. You can even analyze programs that don't compile, a particularly nice feature for those porting code.

The Static Analyzer works by reading through source code files that you specify and creating a database of functions, macros, variables, files, and (for C++) classes and methods and their relationships. The main Static Analyzer window with a typical call graph is illustrated in Figure 5.

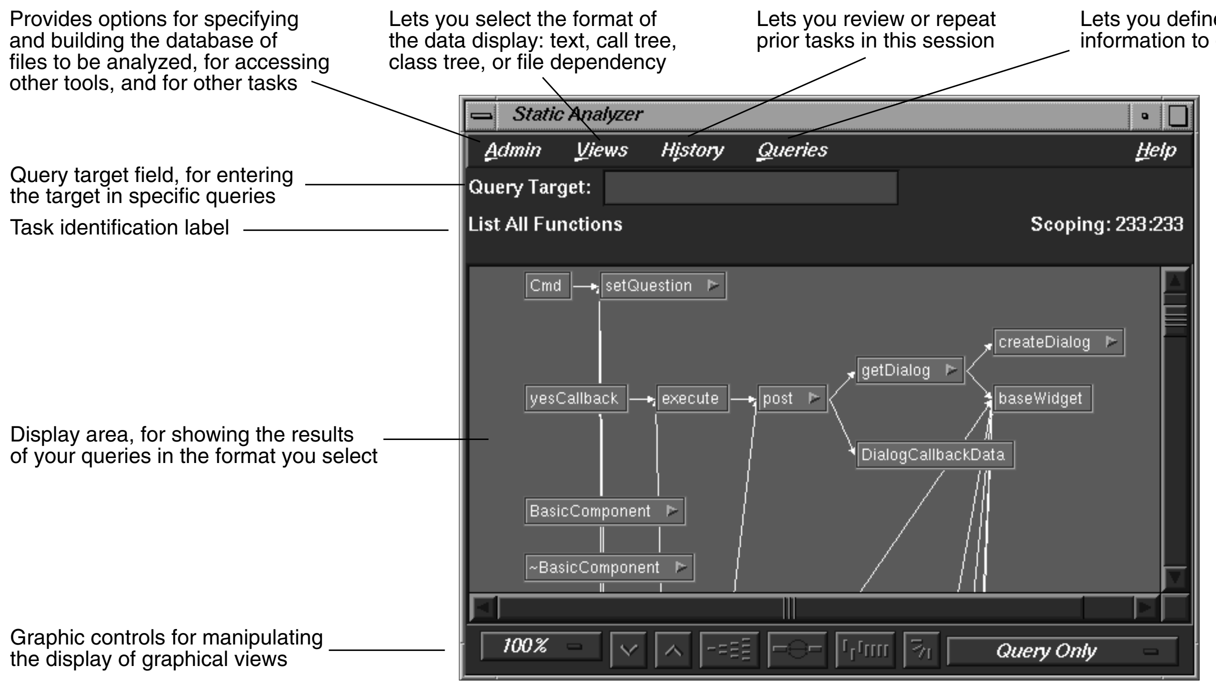

Provides options for specifying and building the database of files to be analyzed, for accessing other tools, and for other tasks
Lets you select the format of the data display: text, call tree, class tree, or file dependency
Lets you review or repeat prior tasks in this session
Lets you define the type of information to display
Query target field, for entering the target in specific queries
Task identification label
Display area, for showing the results of your queries in the format you select
Graphic controls for manipulating the display of graphical views
Static Analyzer
Admin
Views
History
Queries
Help
Query Target:
List All Functions
Scoping: 233:233
Cmd
setQuestion
createDialog
getDialog
yesCallback
execute
post
baseWidget
DialogCallbackData
BasicComponent
~BasicComponent
100%
Query Only

Figure 5 Main Static Analyzer Window

Static Analyzer User Model

Follow these steps for using the Static Analyzer:

1. Invoke the Static Analyzer, either by typing **`cvstatic`** or by selecting "Static Analyzer" from the Launch submenu in any ProDev WorkShop Admin menu (preferably from the directory where your source is located).
2. Decide which files are to be analyzed.

   You designate which files are to be analyzed in a special file called a *fileset*. A fileset is a regular ASCII file with a format of one entry per line, each line separated from the next by a carriage return. The entries can be regular expressions, filenames, or included directories preceded by the designator `-I`.

   To specify a fileset, you can

   - create the fileset manually using a text editor
   - use the Fileset Editor, which is accessed from the Admin menu in the Static Analyzer window
   - let the Static Analyzer create the fileset automatically at startup by defaulting to the files in the current directory that match the expression `*.[cCfF]`
   - let the Static Analyzer create the fileset automatically at startup from the command line by typing **`cvstatic`** with the **-executable** flag and designating an executable
   - use the compiler to create a fileset (and database) by adding the **`-sa,`**<*dbdirectory*> option to your makefile

   Note: Many programs are so big that a query covering the entire scope is useless due to its size and complexity. There are two ways to keep the scope of your analysis at a manageable size: (1) Limit the number of files to be analyzed or (2) avoid queries that begin with "List All ...".
3. Decide how you are going to build the database.

   Before you can specify a fileset, you need to decide how you are going to build the database. You can choose to create the database in *scanner* mode (the default), which is fast but not sensitive to any specific programming language, or in *parser* mode, which uses the compiler and is slower but more thorough. Use scanner mode for large programs or for programs that do not compile. Scanner mode is particularly suited to porting situations. Parser mode is better when you have code that compiles and you need to determine language-specific relationships, particularly in Fortran and C++.

4. Build the database.
5. Perform your queries.

   Queries are selected from the Queries menu in the Static Analyzer. They fall into 10 categories, as shown in Figure 6. Remember that the "List All ..." queries can produce overwhelming results for large programs.
6. View (and save) the results.

   The Static Analyzer has four ways of presenting data, which are selected from the Views menu:

   - "Text View" displays query results in a text format. In addition to listing the queried items, it indicates the source filename and line number, and includes the actual source line.
   - "Call Tree View" applies to function queries. It presents the data in a graphical format with *nodes* (rectangles) representing functions and *arcs* (arrows) representing calls to functions.
   - "Class Tree View" applies to C++ class queries. It presents a class inheritance tree with nodes representing classes and arcs representing parent-child class relationships.
   - "File Dependency View" applies to file queries. It presents a graph, with nodes representing files and arcs representing include relationships.

   If you want to save a query in a graphical view, you can save a PostScript® version by selecting "Save Query..." from the Admin menu and print it out at your leisure.
7. Access the source code.

   Double-clicking any node in a graph or item in Text View brings up the Source View window containing the corresponding source code. Double-clicking any arc (arrow) displays Source View with the corresponding call site or file inclusion.

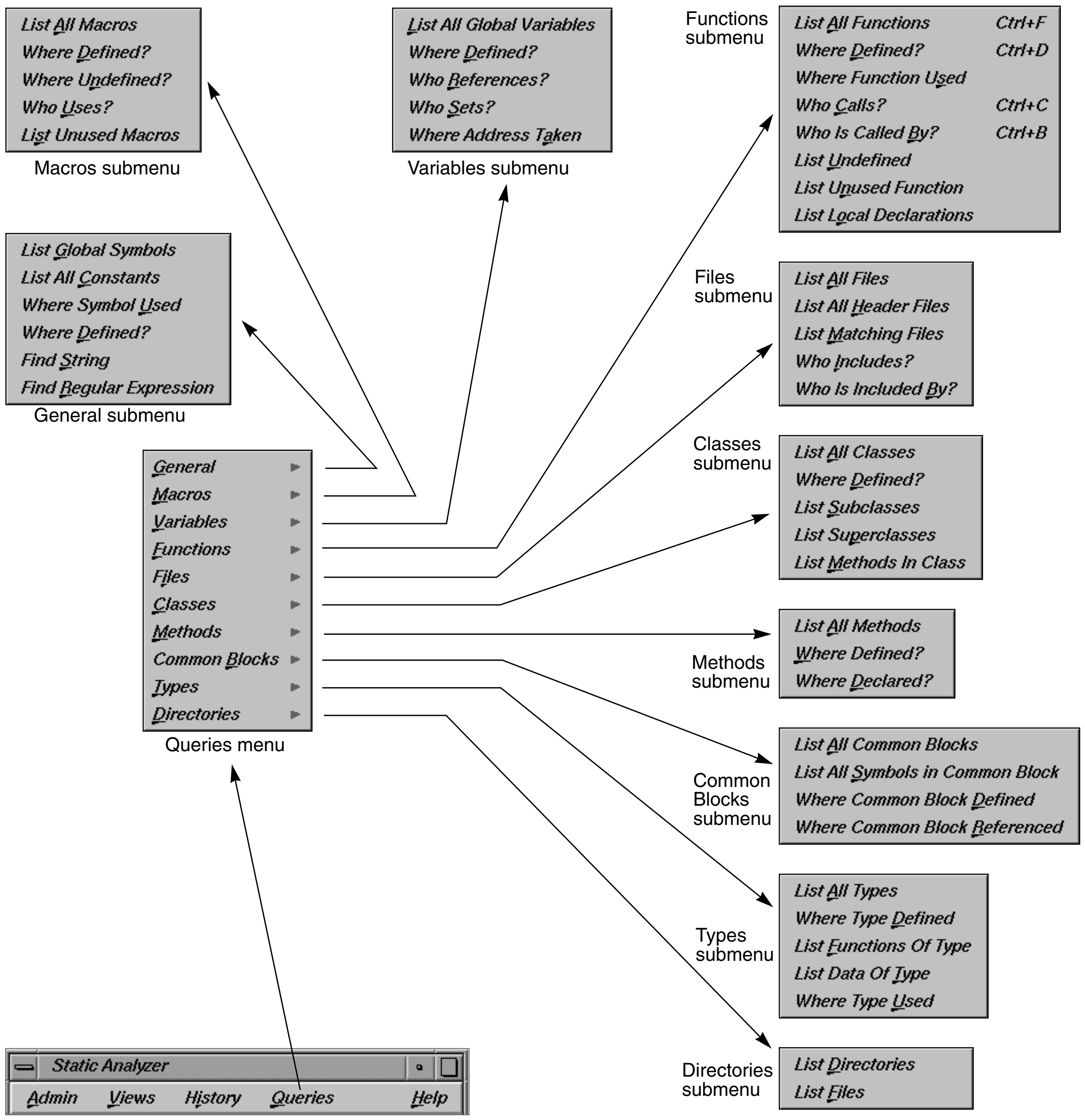


Figure 6 Queries Available from the Static Analyzer Queries Submenus

Where to Find Static Analyzer Information

To find out more about the Static Analyzer, refer to Table 2.

Table 2 Where to Find Static Analyzer Information in the *CASEVision/WorkShop User's Guide*

Topic	See ...
General Static Analyzer description	Chapter 12, "Introduction to the WorkShop Static Analyzer"
Static Analyzer tutorial	Chapter 13, "A Sample Session with the Static Analyzer"
Specifying a fileset	"Fileset Specifications" on page 238
Building a database using scanner mode	"Scanner Mode" on page 247
Building a database using parser mode	"Parser Mode" on page 248
Performing queries	Chapter 15, "Static Analyzer: Queries"
Static Analyzer viewing formats	Chapter 16, "Static Analyzer: Views"
Strategies for analyzing large programs	Chapter 17, "Static Analyzer: Working on Large Programming Projects"

C++ Browser User Model

The C++ Browser user model is similar to the Static Analyzer user model. After you build the database (which must be done in parser mode), you access the C++ Browser by selecting "C++ Browser" from the Static Analyzer Admin menu.

The C++ Browser lets you display different sets of information about classes, class members, and interclass relationships through these three views:

- Class View— displays member and class information in an expandable, hierarchical outline format with the members of the current class in the left pane and related classes on the right (see Figure 7). Clicking the diamond-shaped icons next to the headings in the list hides or displays the associated information.

Like the Static Analyzer, you have numerous queries available through the Query menu. In addition, if you select an item in either of the Class View lists and hold down the right mouse button, you can access the Queries menu specific to that type of item, that is, methods, data members, classes, and so on.

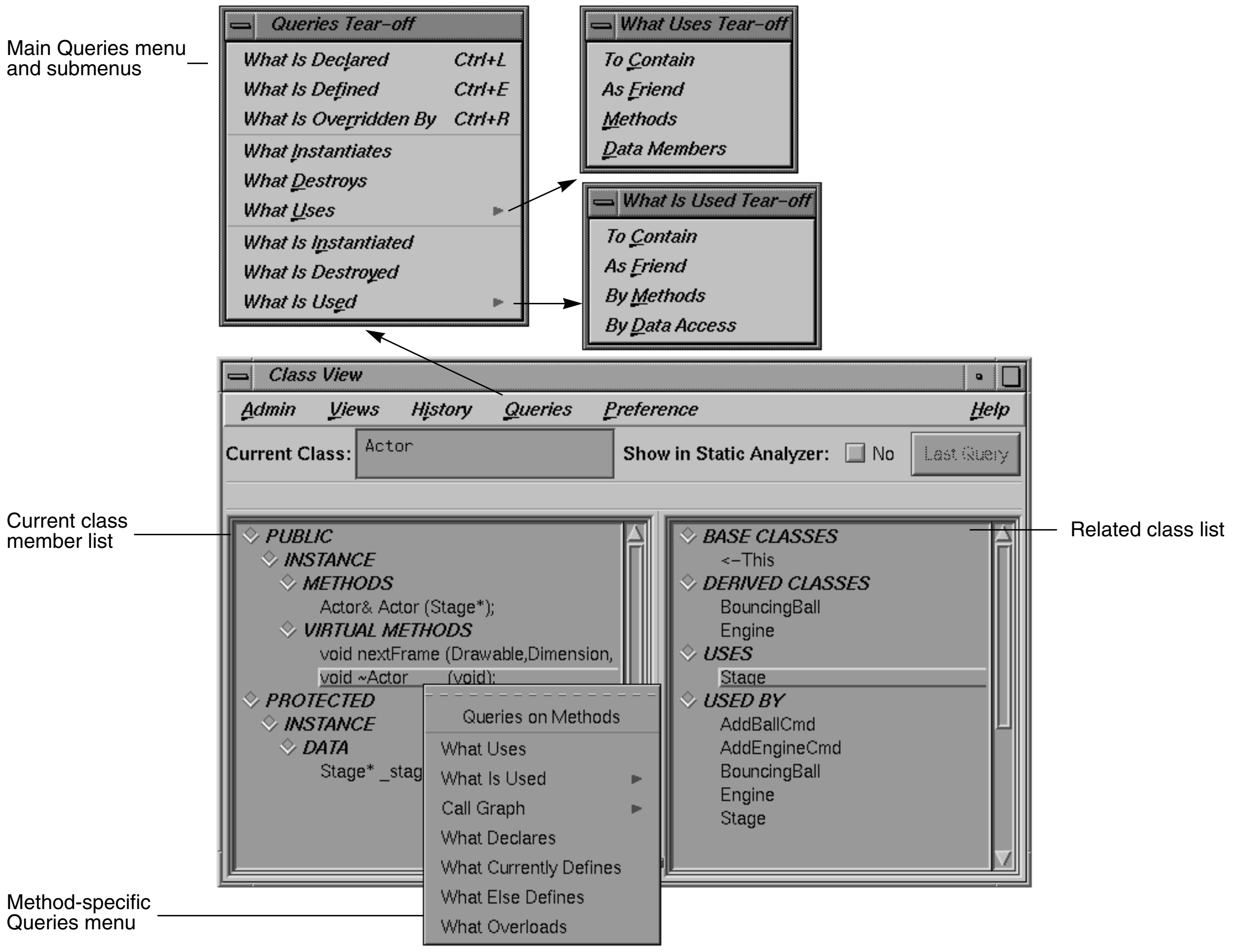


Figure 7 C++ Browser Class View Window with Query Menus

You can also create man pages describing classes by selecting "Generate man pages..." from the Class View Admin menu. You simply specify an individual class or all of them and the C++ Browser fills in the man page template for you.

- Class Graph—displays the class hierarchy for the current class in the Class View window with nodes as classes and arcs as parent-child relationships. Class Graph can show four types of relationships: inheritance, containment, interaction, and friends. You can display all classes, limit the scope of classes derived from the current class, or get a butterfly view showing the immediate base and derived classes of the current class.
- Call Graph—displays the calling relationships of methods and virtual methods selected from Class View with options for customizing the display of the graph.

To find out more about the C++ Browser, refer to Table 3.

Table 3 Where to Find C++ Browser Information in the *CASEVision/WorkShop MegaDev User's Guide*

Topic	**See ...**
General C++ Browser description	Chapter 5, "Getting Started with the C++ Browser"
C++ Browser tutorial	Chapter 6, "Using the C++ Browser: A Sample Session"
Class View	"Using the Class View Outline Lists" on page 83 and "Class View Window" on page 117
Viewing graph data	"Class Graph and Call Graph Displays" on page 147
Call Graph	"Using the Call Graph Window" on page 107 and "Call Graph Window" on page 156
Class Graph	"Using the Class Graph Window" on page 101 and "Class Graph Window" on page 154

Pinpointing Performance Problems With the Performance Analyzer

The ProDev WorkShop Performance Analyzer helps you understand how your program performs so that you can correct any problems. In performance analysis, you run experiments to capture performance data and see how long each phase or part of your program takes to run. You can then determine if the performance of the phase is slowed down by the CPU, I/O activity, memory, or a bug and attempt to speed it up.

A menu of predefined tasks is provided to help you set up your experiments. With the Performance Analyzer views, you can conveniently analyze the data. These views show CPU utilization and process resource usage (such as context switches, page faults, and working set size), I/O activity, and memory usage (to capture such problems as memory leaks, bad allocations, and heap corruption).

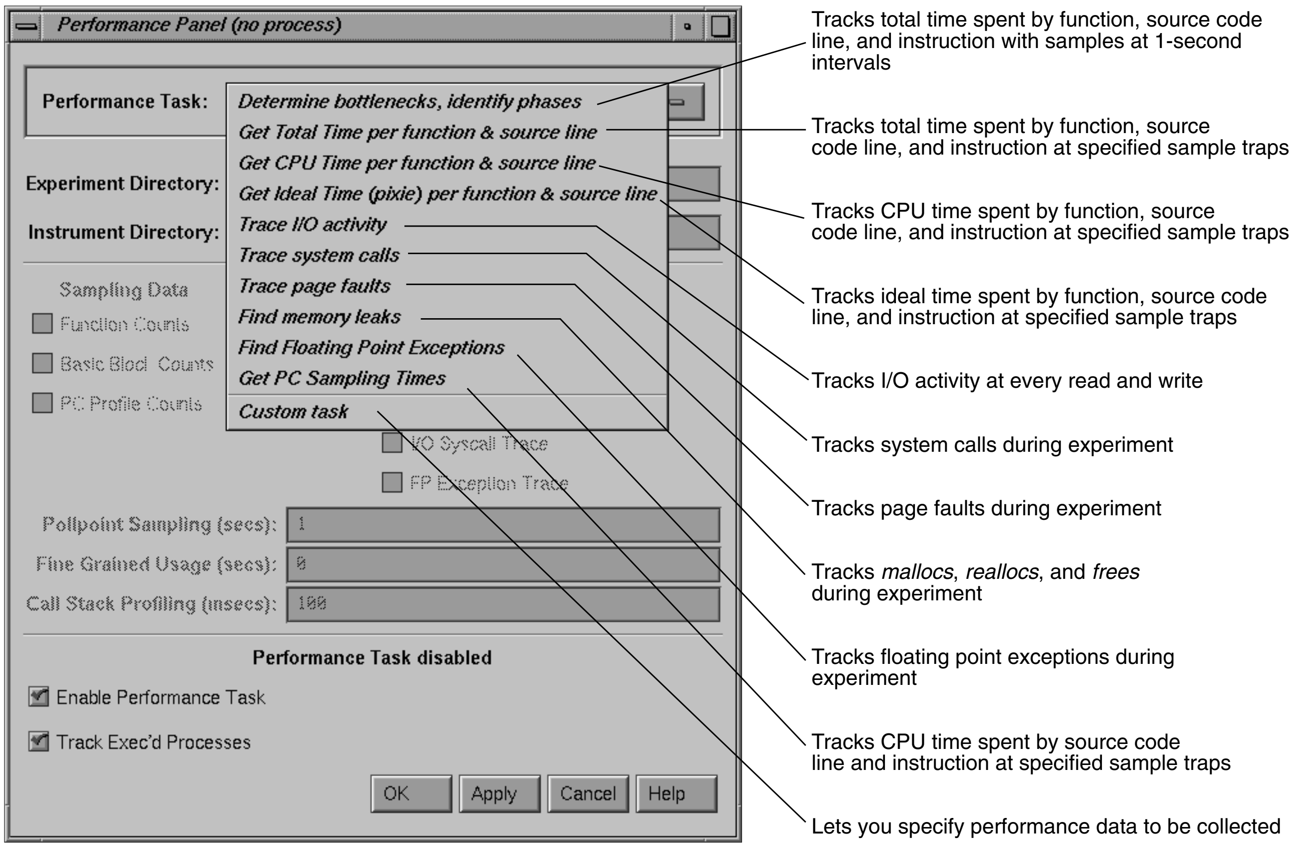
The Performance Analyzer has three general techniques for collecting performance data:

- Counting—It can count the exact number of times each function and/or basic block has been executed. This requires *instrumenting* the program, that is, inserting code into the executable to collect counts.
- Profiling—It can periodically examine and record the program's PC (program counter), call stack, and resource consumption.
- Tracing—It can trace events that affect performance, such as *reads* and *writes*, system calls, page faults, floating point exceptions, and *mallocs*, *reallocs*, and *frees*.

Performance Analyzer User Model

1. Set up a general experiment to determine areas for improvement in your program.

   To set up a performance experiment, select "Performance Task..." from the Admin menu in the Debugger Main View or type **`cvspeed`** at the command line. The Performance Panel window is displayed; it provides a task menu from which you select predefined experiment tasks (see Figure 8). At this point, you probably haven't formed a hypothesis yet about where the performance problems lie. If this is the case, select the "Determine bottlenecks, identify phases" task. This is useful for determining the general problem areas within the program.


Performance Panel (no process)
Performance Task:
Determine bottlenecks, identify phases
Get Total Time per function & source line
Get CPU Time per function & source line
Get Ideal Time (pixie) per function & source line
Trace I/O activity
Trace system calls
Trace page faults
Find memory leaks
Find Floating Point Exceptions
Get PC Sampling Times
Custom task
Experiment Directory:
Instrument Directory:
Sampling Data
Function Counts
Basic Block Counts
PC Profile Counts
I/O Syscall Trace
FP Exception Trace
Performance Task disabled
Enable Performance Task
Track Exec'd Processes
OK
Apply
Cancel
Help
Tracks total time spent by function, source code line, and instruction with samples at 1-second intervals
Tracks total time spent by function, source code line, and instruction at specified sample traps
Tracks CPU time spent by function, source code line, and instruction at specified sample traps
Tracks ideal time spent by function, source code line, and instruction at specified sample traps
Tracks I/O activity at every read and write
Tracks system calls during experiment
Tracks page faults during experiment
Tracks *mallocs*, *reallocs*, and *frees* during experiment
Tracks floating point exceptions during experiment
Tracks CPU time spent by source code line and instruction at specified sample traps
Lets you specify performance data to be collected

Pollpoint Sampling (secs): 1
Fine Grained Usage (secs): 0
Call Stack Profiling (msecs): 100

Figure 8 Performance Panel With Task Menu Displayed

2. Start the program by clicking the *Run* button in Main View.

   This runs the experiment and collects the performance data, which is written to a directory *test0000* (or a name of your choice); *test0000* is the identification for your experiment.

3. Analyze the results in the Performance Analyzer window and the Usage View (Graphs) window.

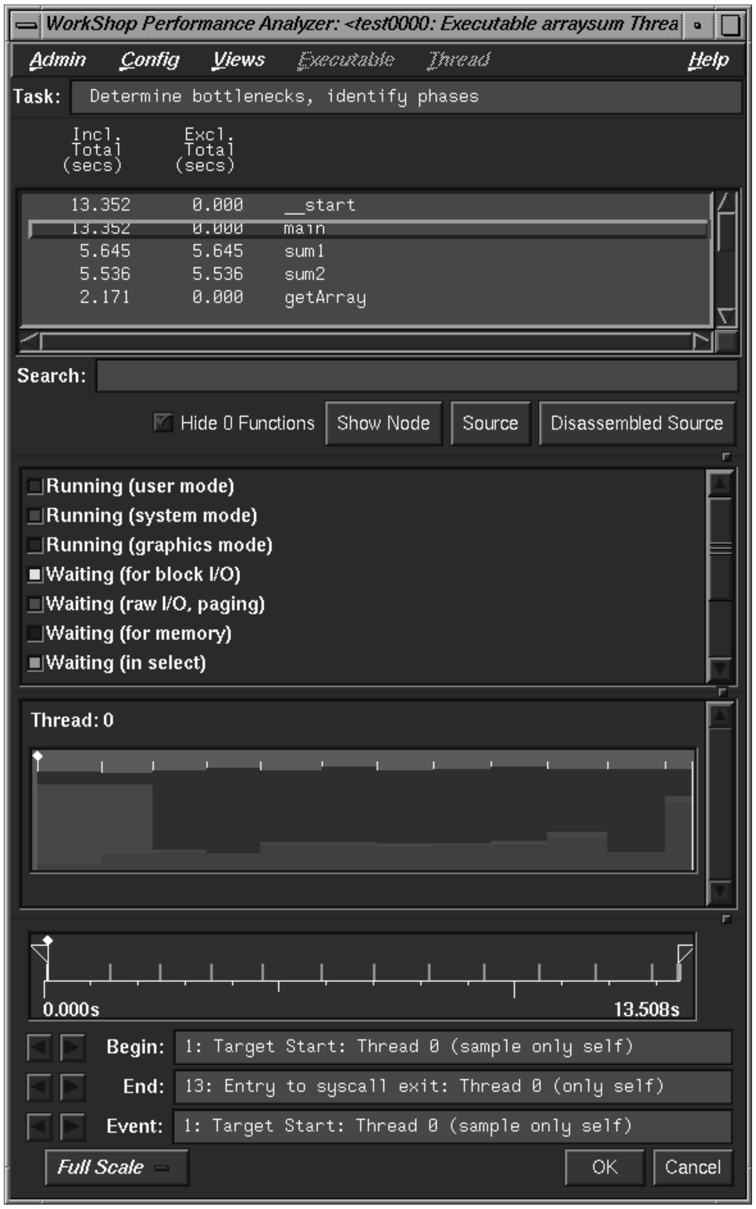
After the experiment has finished, you can display the results in the Performance Analyzer window by selecting "Performance Analyzer" from the Launch submenu in any ProDev WorkShop Admin menu or by typing **`cvperf -exp`** *experimentname*. The results from a typical performance analysis experiment appear in Figure 9, the main Performance Analyzer window, and Figure 10, which shows a subset of the graphs in the Usage Views (Graphs) window. You should be able to determine where the phases of execution occur so that you can set *sample traps* between them. Sample traps collect performance data at specified times and events in the experiment.

Function list area displays functions with their performance data from the experiment, time spent in the function including its called functions, and time spent in the function excluding calls

Usage chart area indicates general resource usage during the experiment

Time line area indicates where samples were taken. The calipers let you limit the scope of the analysis to a portion of the experiment. Double-clicking a sample point displays the call stack that occurred there.

Caliper and sample point selector controls



Task: Determine bottlenecks, identify phases

```
  Incl.      Excl.
  Total      Total
  (secs)     (secs)
 13.352      0.000    __start
 13.352      0.000    main
  5.645      5.645    sum1
  5.536      5.536    sum2
  2.171      0.000    getArray
```

Begin: 1: Target Start: Thread 0 (sample only self)

End: 13: Entry to syscall exit: Thread 0 (only self)

Event: 1: Target Start: Thread 0 (sample only self)

Figure 9 Performance Analyzer Main Window

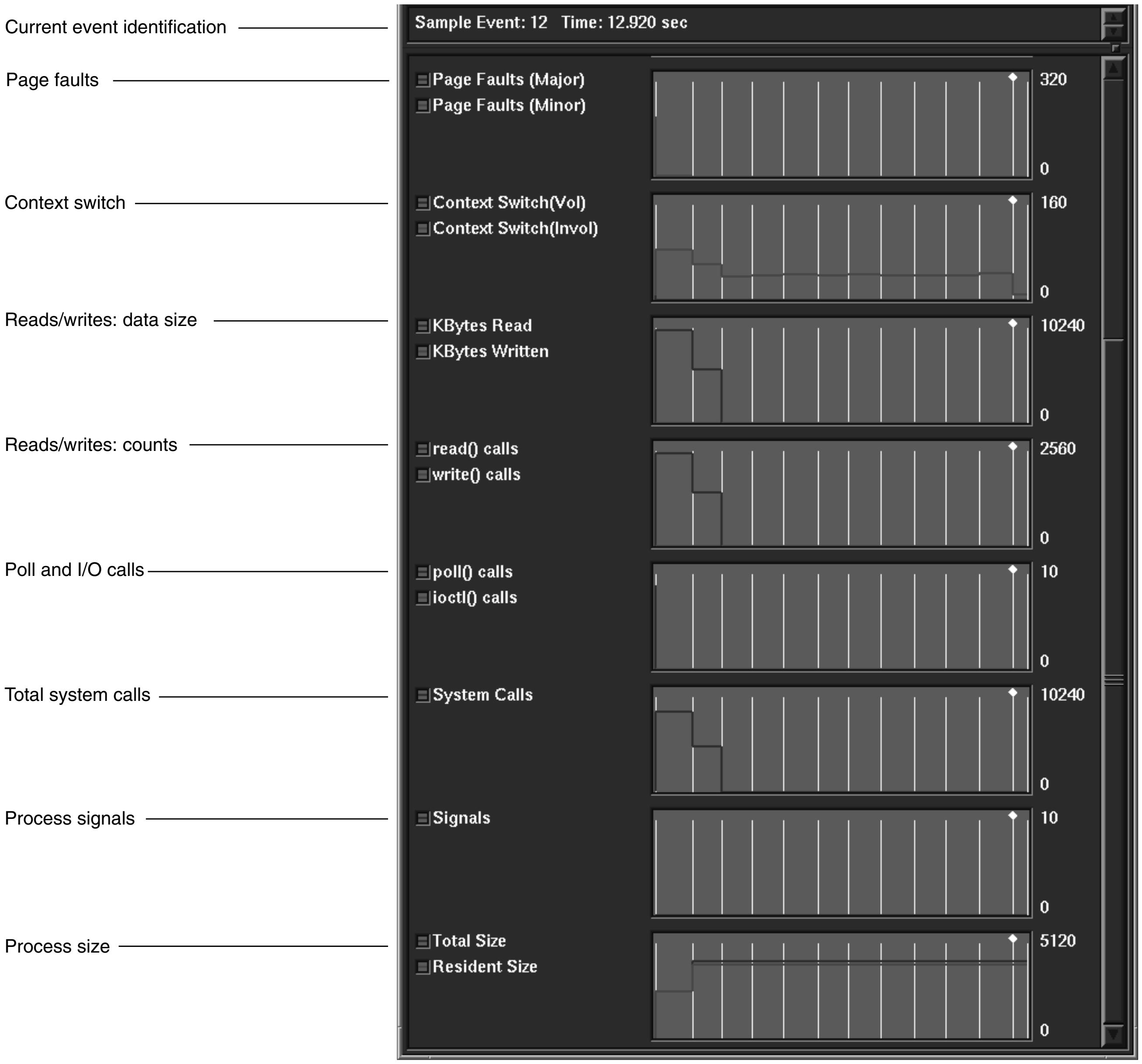


Figure 10 Usage View (Graphs) Window: Lower Graphs

4. Set sample traps at the start and end of each phase.

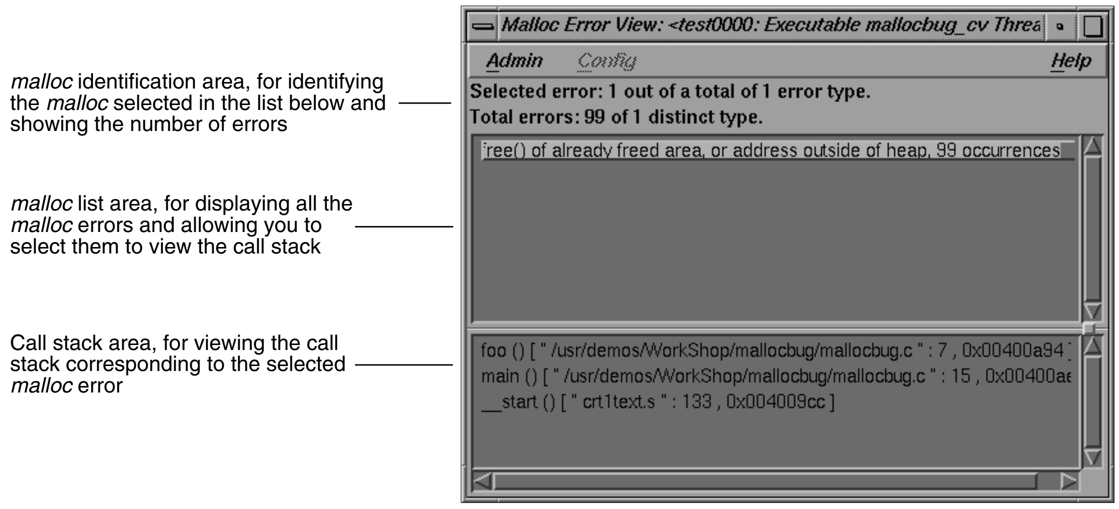
   Setting sample traps between phases isolates the data to be analyzed on a phase-by-phase basis. Sample traps are set by selecting “Sample”, “Sample at Function Entry”, or “Sample at Function Exit” from the Set Trap submenu in the Traps menu in the Debugger Main View or through the Traps Manager.

5. Select your next experiment from the Task Menu in the Performance Panel and run it by clicking the *Run* button in the Main View window.

   You need to form a hypothesis about the performance problem and select an appropriate task (see Figure 8) for your next experiment. There are trade-offs in selecting tasks—experiments can collect huge amounts of data and may perturb the results in some cases. The strategies for selecting tasks are discussed in detail in Chapter 20, “Setting Up Performance Analysis Experiments,” in the *CASEVision/WorkShop User’s Guide*.

6. Analyze the results using the Performance Analyzer main window, its views, or Source View with performance data annotations displayed.

   A typical Performance Analyzer view, Malloc Error View, is shown in Figure 11. The Performance Analyzer provides results in the windows listed in Table 4.


Malloc Error View: <test0000: Executable mallocbug_cv Threa
Admin
Config
Help
malloc identification area, for identifying the *malloc* selected in the list below and showing the number of errors
malloc list area, for displaying all the *malloc* errors and allowing you to select them to view the call stack
Call stack area, for viewing the call stack corresponding to the selected *malloc* error

```
Selected error: 1 out of a total of 1 error type.
Total errors: 99 of 1 distinct type.
ree() of already freed area, or address outside of heap, 99 occurrences
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 7 , 0x00400a94
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0x00400ae
__start () [ " crt1text.s " : 133 , 0x004009cc ]
```

Figure 11 Malloc Error View

Table 4 Performance Analyzer Views and Data

Performance Analyzer Window	**Data Provided**
Performance Analyzer main window	Function list with performance data, usage chart showing general resource usage over time, and time line for setting scope on data
Call Stack View	Call stack recorded when selected event occurred
Usage View (Graphs)	Specific resource usage over time, shown as graphs
Usage View (Numerical)	Specific resource usage for selected (by caliper) time interval, shown as numerical values
Call Graph View	A graph showing functions that were called during the time interval, annotated by the performance data collected
I/O View	A graph showing I/O activity over time during the time interval
Malloc View	A list of all *mallocs*, their sizes and number of occurrences, and, if selected, their corresponding call stack within the selected time interval
Malloc Error View	A list of *malloc* errors, their number of occurrences, and if selected, their corresponding call stack within the time interval
Leak View	A list of specific leaks, their sizes and number of occurrences, and if selected, their corresponding call stack within the time interval
Heap View	A generalized view of heap memory within the time interval
Source View	The ProDev WorkShop text editor window showing source code annotated by performance data collected
Working Set View	The instruction coverage of dynamic shared objects (DSOs) that make up the executable, showing instructions, functions, and pages that were not used within the time interval
Cord Analyzer	The Cord Analyzer is not actually part of the Performance Analyzer, but it works with data from Performance Analyzer experiments. It lets you try out different ordering of functions to see the effect on performance.

Where to Find Performance Analyzer Information

To find out more about the Performance Analyzer, refer to Table 5.

Table 5 Where to Find Performance Analyzer Information in the *CASEVision/WorkShop User's Guide*

Topic	See ...
General Performance Analyzer information	Chapter 18, "Introduction to the Performance Analyzer"
Performance analysis theory	"Sources of Performance Problems" on page 312
General Performance Analyzer tutorial	Chapter 19, "Performance Analyzer Tutorial"
Memory leak tutorial	"Memory Experiment Tutorial" on page 415
Setting up performance analysis experiments including task selection	Chapter 20, "Setting Up Performance Analysis Experiments" for details and "The Performance Panel" on page 362 for a summary
Setting sample traps	"Setting Traps in Main View and Source View" on page 73 and "Setting Traps in Trap Manager" on page 76
Performance Analyzer main window	"The Performance Analyzer Main Window" on page 374
Usage View (Graphs) window	"Usage View (Graphs)" on page 387
Watching an experiment without collecting data in the Process Meter	"Process Meter" on page 393
Usage View (Numerical) window	"Usage View (Numerical)" on page 393
Tracing I/O calls using the I/O View window	"I/O View" on page 396
Call Graph View window	"Call Graph View" on page 397
Finding memory problems	"Analyzing Memory Problems" on page 405
Specifying performance annotations for Source View and Call Graph View	"Config Menu" on page 382
Call Stack View window	"Call Stack" on page 418
Improving working set behavior	"Analyzing Working Sets" on page 419

Determining the Thoroughness of Test Coverage With Tester

Tester is a software quality assurance toolset for measuring dynamic coverage over a set of tests. It tracks the execution of functions, individual source lines, arcs, blocks, and branches.

Tester User Model

This section describes the user model for designing a single test. After you have your instrumentation file and test directories set up, you can automate your testing and create larger test sets. Tester has both a command line interface (see Table 6) and a graphical user interface (see Figure 12).

1. Plan your test.
2. Create (or reuse) an instrumentation file.

   The instrumentation file defines the coverage data you wish to collect in this test.
3. Apply the instrument file to the target executable(s).

   This creates a special executable for testing purposes that collects data as it runs.
4. Create a test directory to collect the data files.
5. Run the instrumented version of the executable to collect the coverage data.
6. Analyze the results.

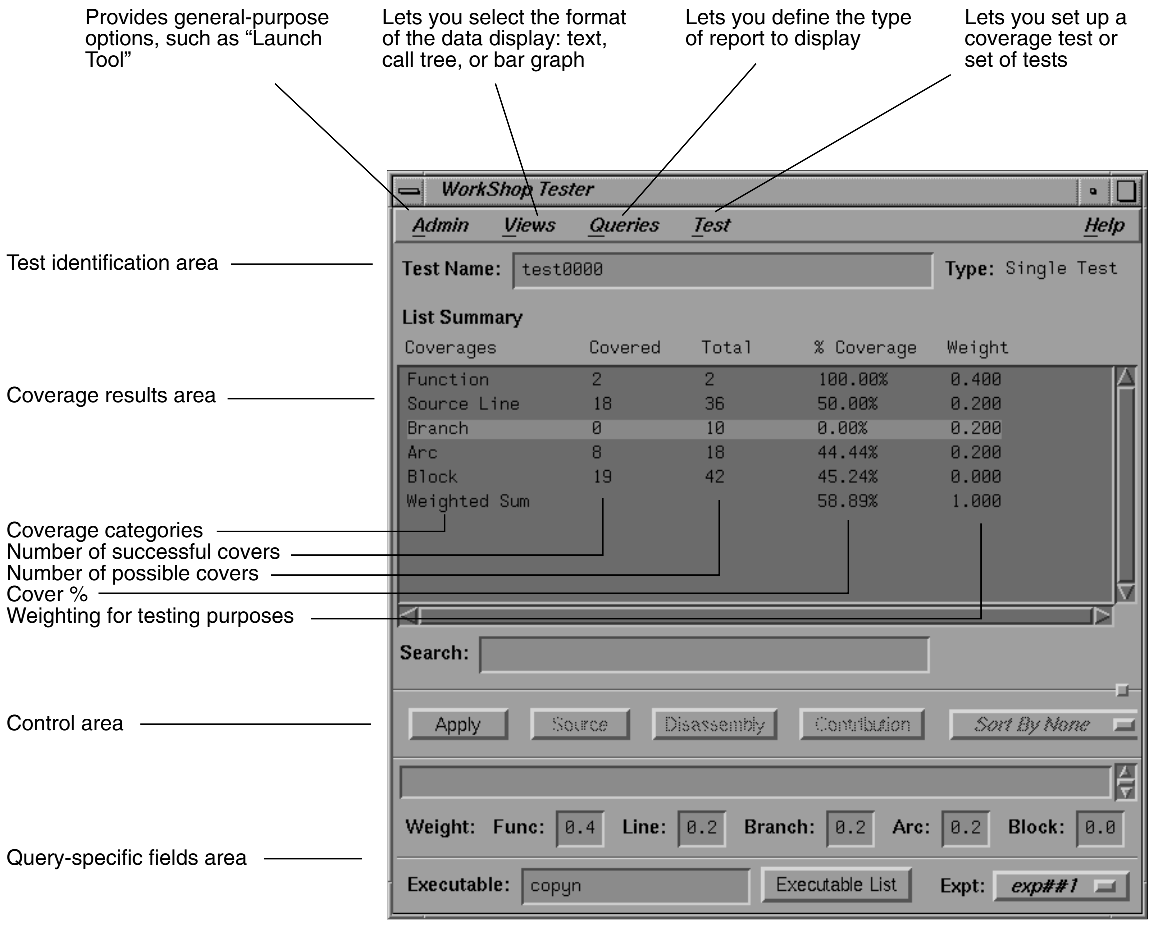
   Tester produces a wide variety of column-based reports. Most are available in both interfaces: command line and graphical. The reports can show source and assembly line coverage; coverage of functions; arc coverage, that is, coverage of function calls; call graphs indicating caller and callee functions and their counts; basic block counts; count information for assembly language branches; summaries of overall coverage; and argument tracing.

Table 6 Tester Command Line Interface Summary

Command Category	Command Name	Description
general	cvcov cattest	Describes the test details for a test, test set, or test group
	cvcov lsinstr	Displays the instrumentation information for a particular test
	cvcov lstest	Lists the test directories in the current working directory
	cvcov mktest	Creates a test directory
	cvcov rmtest	Removes tests and test sets
	cvcov runinstr	Adds code to the target executable to enable you to capture coverage data, according to the criteria you specify
	cvcov runtest	Runs a test or a set of tests
coverage analysis	cvcov lssum	Shows the overall coverage based on the user-defined weighted average over function, line, block, branch, and arc coverage
	cvcov lsfun	Lists coverage information for the specified functions in the program that was tested
	cvcov lsblock	Displays a list of blocks for one or more functions and the count information associated with each block
	cvcov lsbranch	Lists coverage information for branches in the program, including the line number at which the branch occurs
	cvcov lsarc	Shows arc coverage, that is, the number of arcs taken out of the total possible arcs
	cvcov lscall	Lists the call graph for the executable with counts for each function
	cvcov lsline	Lists the coverage for native source lines
	cvcov lssource	Displays the source annotated with line counts
	cvcov lstrace	Shows the argument tracing information
	cvcov diff	Shows the difference in coverage for different versions of the same program
test set	cvcov mktset	Makes a test set
	cvcov addtest	Adds a test or test set to a test set or test group
	cvcov deltest	Removes a test or test set from a test set or test group

Table 6 **(continued)** Tester Command Line Interface Summary

Command Category	**Command Name**	**Description**
	cvcov optimize	Selects the minimum set of tests that give the same coverage or meet the given coverage criteria as the given set
test group	cvcov mktgroup	Creates a test group that can contain other tests or test groups; targets are either the target libraries or DSOs


Provides general-purpose options, such as “Launch Tool”
Lets you select the format of the data display: text, call tree, or bar graph
Lets you define the type of report to display
Lets you set up a coverage test or set of tests
Test identification area
Coverage results area
Coverage categories
Number of successful covers
Number of possible covers
Cover %
Weighting for testing purposes
Control area
Query-specific fields area
WorkShop Tester
Admin
Views
Queries
Test
Help
Apply
Source
Disassembly
Contribution
Sort By None
Executable List

Test Name: test0000 Type: Single Test

List Summary

```
Coverages          Covered    Total      % Coverage   Weight
Function           2          2          100.00%      0.400
Source Line        18         36         50.00%       0.200
Branch             0          10         0.00%        0.200
Arc                8          18         44.44%       0.200
Block              19         42         45.24%       0.000
Weighted Sum                             58.89%       1.000
```

Search:

Weight: Func: 0.4 Line: 0.2 Branch: 0.2 Arc: 0.2 Block: 0.0

Executable: copyn Expt: exp##1

Figure 12 Major Areas of the Main Tester Window

Where to Finder Tester Information

To find out more information about Tester, refer to Table 7.

Table 7 Where to Find Tester Information in the *CASEVision/WorkShop User's Guide*

Topic	**See ...**
General Tester information	"Tester Overview" on page 435
Automated testing	"Automated Testing" on page 446
Command line interface tutorial	Chapter 23, "Tester Command Line Interface Tutorial"
Graphical user interface tutorial	Chapter 25, "Tester Graphical User Interface Tutorial"
Command line interface details	Chapter 24, "Tester Command Line Reference"
Graphical user interface details	Chapter 26, "Tester Graphical User Interface Reference"

Recompiling Within the ProDev WorkShop Environment With Build Manager

The Build Manager lets you view file dependencies and compiler requirements, fix compile errors conveniently, and compile software without leaving the WorkShop environment. It provides two views:

- Build View—for compiling, viewing compile error lists, and accessing the code containing the errors in Source View (the ProDev WorkShop editor) or an editor of your choice.
- Build Analyzer—for viewing build dependencies and recompilation requirements and accessing source files.

For more information on Build Manager, see Chapter 11, "Using the Build Manager," in the *CASEVision/WorkShop User's Guide*.

Making Quick Changes With Fix and Continue

Fix and Continue is part of the Developer Magic MegaDev module. The Fix and Continue feature lets you make minor changes to your code from within WorkShop without having to recompile and link the entire system. You issue Fix and Continue commands in the Debugger Main View window, either by selecting them from the Fix+Continue menu or typing them in directly in the Debugger command line area.

With Fix and Continue, you can edit a function, parse the new function, and continue execution of the program being debugged. Fix and Continue enables you to speed up your development cycle significantly. For example, a program that takes 5 minutes to rebuild through a conventional compile might take 45 seconds using Fix and Continue.

Fix and Continue lets you:

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

Figure 13 shows you the WorkShop Main View during a Fix and Continue session and explains how to use the Fix and Continue menu.

Fix and Continue User Model

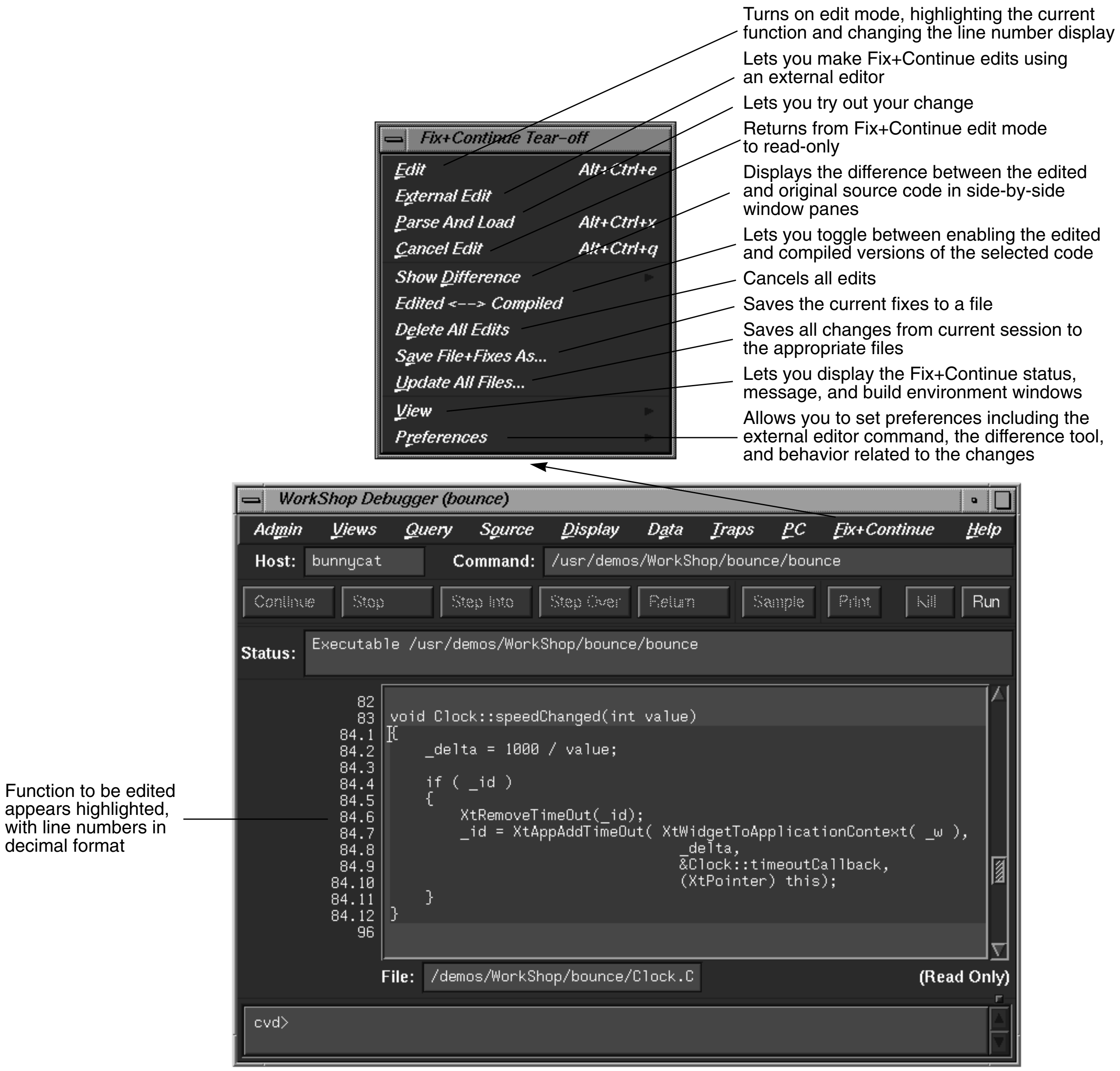
1. Invoke the Debugger as you normally would by typing:

   `cvd` `[`**`-pid`** *pid*`]` `[`**`-host`** *host*`]` `[`*executable* `[`*corefile*`]]` `[`**`&`**`]`

   See "Debugger User Model" on page 4.

2. Navigate to the function to be changed.

   You can get to the function numerous ways, by selecting "Search..." from the Source menu, typing **`func`** *functionname* at the Debugger command line, or simply scrolling to the location. If you did not use **`func`** *functionname*, you need to place the cursor inside the function.



Turns on edit mode, highlighting the current function and changing the line number display

Lets you make Fix+Continue edits using an external editor

Lets you try out your change

Returns from Fix+Continue edit mode to read-only

Displays the difference between the edited and original source code in side-by-side window panes

Lets you toggle between enabling the edited and compiled versions of the selected code

Cancels all edits

Saves the current fixes to a file

Saves all changes from current session to the appropriate files

Lets you display the Fix+Continue status, message, and build environment windows

Allows you to set preferences including the external editor command, the difference tool, and behavior related to the changes

Host: bunnycat

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

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

```
   82
   83  void Clock::speedChanged(int value)
 84.1  {
 84.2      _delta = 1000 / value;
 84.3
 84.4      if ( _id )
 84.5      {
 84.6          XtRemoveTimeOut(_id);
 84.7          _id = XtAppAddTimeOut( XtWidgetToApplicationContext( _w ),
 84.8                                 _delta,
 84.9                                 &Clock::timeoutCallback,
84.10                                 (XtPointer) this);
84.11      }
84.12  }
   96
```

File: /demos/WorkShop/bounce/Clock.C (Read Only)

```
cvd>
```

Function to be edited appears highlighted, with line numbers in decimal format

Figure 13 Using Fix+Continue

3. Select “Edit” from the Fix+Continue menu.

   This turns on edit mode, highlighting the function source code. If line numbers are displayed, those in the selected function appear with a two-part number separated by a decimal point. The left part represents the starting line number of the function in the source file before selecting “Edit”. The right part renumbers the source within the function to make it easier to keep track of added new lines.

4. Make your changes to the source code.

   You can do this directly in Main View or you can use a preferred editor by selecting “External Edit” from the Fix+Continue menu.

5. Try out your changes.

   Selecting “Parse And Load” adds your changes to the executable you are debugging. The changed function will get executed the next time it is invoked. If you stopped in the edited function, the Debugger will let you continue from the corresponding line in the new function, barring certain restrictions.

6. If the changes are satisfactory, save them for later compiling.

   “Save File+Fixes As ...” saves the fixes in your current file. “Update All Files...” saves all fixes in your current session.

At any point, you can make comparisons with your old code. “Show Difference” displays the old and new source code in a side-by-side format. “Edited<-->Compiled” lets you toggle between the old and new executables making it easy to verify or demonstrate your bug fix.

Where to Find Fix and Continue Information

To find out more information about Fix and Continue, refer to Table 8.

Table 8 Where to Find Fix and Continue Information in the *CASEVision/WorkShop MegaDev User's Guide*

Topic	See ...
General information	Chapter 2, "Getting Started with Fix and Continue"
Fix and Continue tutorial	Chapter 3, "Using Fix and Continue: A Sample Session"
Detailed command information	Chapter 4, "Fix and Continue Reference"

Debugging IRIX IM Programs

The IRIX IM Analyzer provides special debugging support for IRIX IM applications and is available from the WorkShop Views menu. The IRIX IM Analyzer operates in a number of modes (referred to as *examiners*) for examining different types of IRIX IM objects. The IRIX IM Analyzer provides information unavailable through conventional debuggers. It also lets you set widget-level breakpoints and collect X event history.

When you first invoke the IRIX IM Analyzer, it comes up in its Widget Examiner mode. You can switch to the other examiners through the Examiner menu or by clicking the tabs at the bottom of the window (See Figure 14).

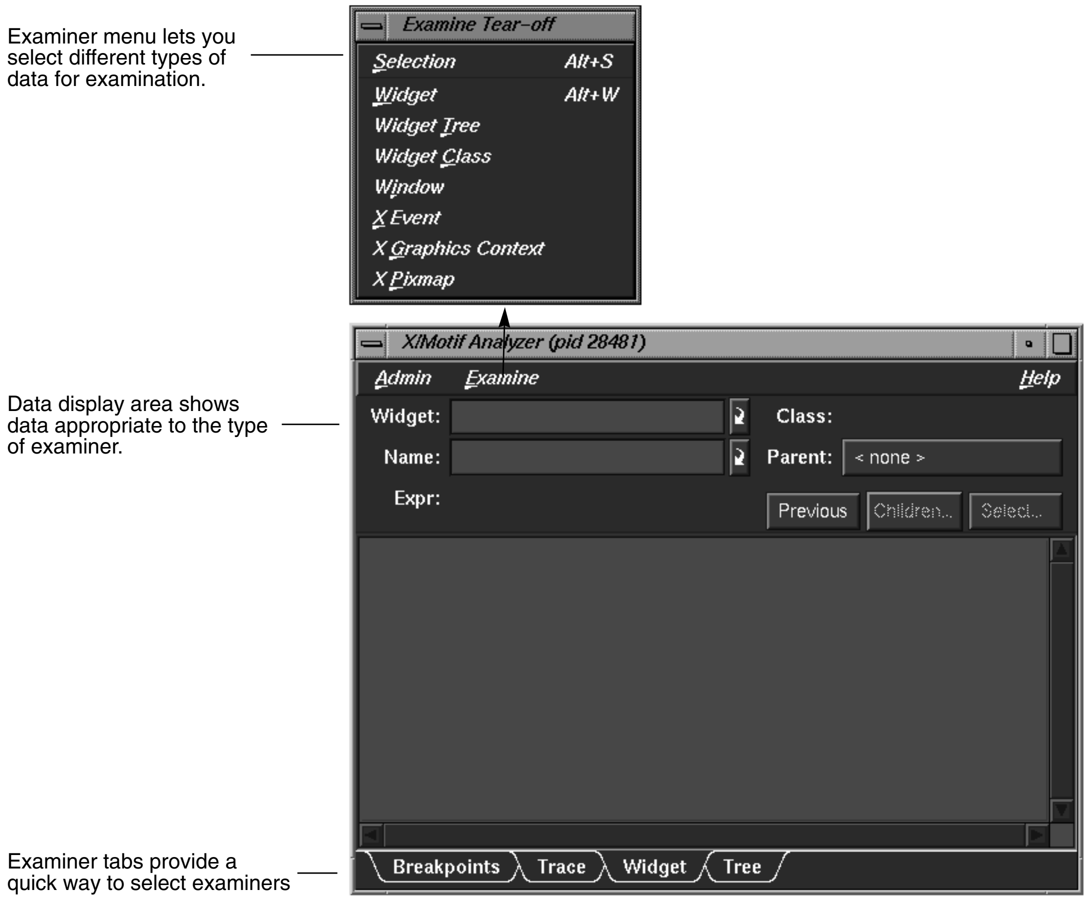

Examiner menu lets you select different types of data for examination.
Data display area shows data appropriate to the type of examiner.
Examiner tabs provide a quick way to select examiners
Examine Tear-off
Selection Alt+S
Widget Alt+W
Widget Tree
Widget Class
Window
X Event
X Graphics Context
X Pixmap
X/Motif Analyzer (pid 28481)
Admin
Examine
Help
Widget:
Class:
Name:
Parent:
< none >
Expr:
Previous
Children...
Select...
Breakpoints
Trace
Widget
Tree

Figure 14 The IRIX IM Analyzer Window

Features of the IRIX IM Analyzer

The IRIX IM Analyzer provides the following types of examiners:

- Widget examiner—identifies a widget's ID, name, class, and parent, and displays its definitions.
- Widget tree examiner—displays the widget hierarchy (see Figure 15). The widgets can be displayed by name, class, or ID by selecting from

the widget display menu. Double-clicking a widget node switches to the widget examiner and displays the data for the selected widget.

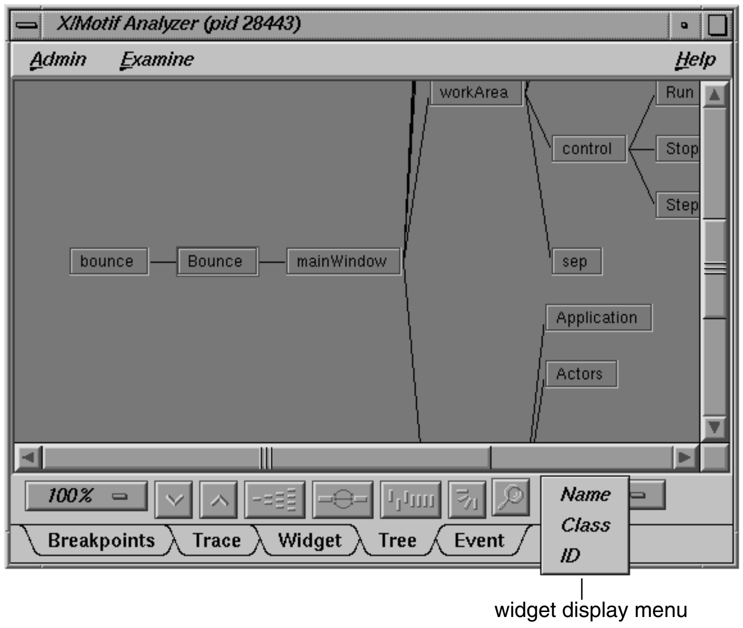
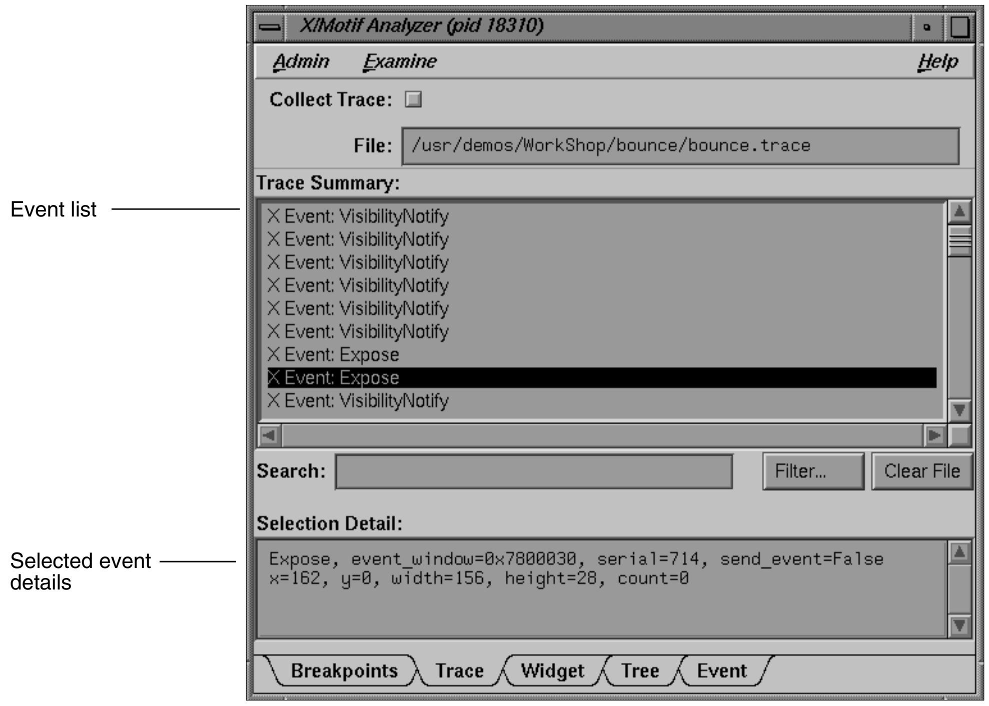


Figure 15 IRIX IM Analyzer Widget Tree Examiner

- Breakpoints examiner—lets you set breakpoints at the widget and widget class level. You can set breakpoints at
  - callback functions
  - widget events
  - resource changes caused
  - timeout callback functions
  - input callback functions
  - widget state changes
  - X events
  - X requests

- Trace examiner—lets you trace the execution of your application and collect the following types of data:
  - X Server Events
  - X Server Requests
  - widget event dispatch information
  - widget resource changes
  - widget state changes
  - Xt callbacks

  Figure 16 is a typical example of the trace examiner. The events appear in a list. Double-clicking an event displays its details.
- Callback examiner—comes up automatically when the process stops in a callback. It displays
  - the callstack frame for the callback function
  - widget information
  - the callback data structure
- Window examiner—identifies the window, its parent and any children, and displays window attribute information
- Event examiner—displays the event structure for a given XEvent pointer
- Graphics context (GC) examiner—displays the X graphics context attributes for a given GC pointer
- Pixmap examiner—displays the basic attributes of an X pixmap, including size and depth, and can provide an ASCII display of small pixmaps, using the units digit of the pixel values
- widget class examiner—displays the widget class attributes for a given widget class pointer



```
/usr/demos/WorkShop/bounce/bounce.trace
```

```
X Event: VisibilityNotify
X Event: VisibilityNotify
X Event: VisibilityNotify
X Event: VisibilityNotify
X Event: VisibilityNotify
X Event: VisibilityNotify
X Event: Expose
X Event: Expose
X Event: VisibilityNotify
```

```
Expose, event_window=0x7800030, serial=714, send_event=False
x=162, y=0, width=156, height=28, count=0
```

Figure 16 IRIX IM Analyzer Trace Examiner

Where to Find IRIX IM Analyzer Information

To find out more information about the IRIX IM Analyzer, refer to Table 9.

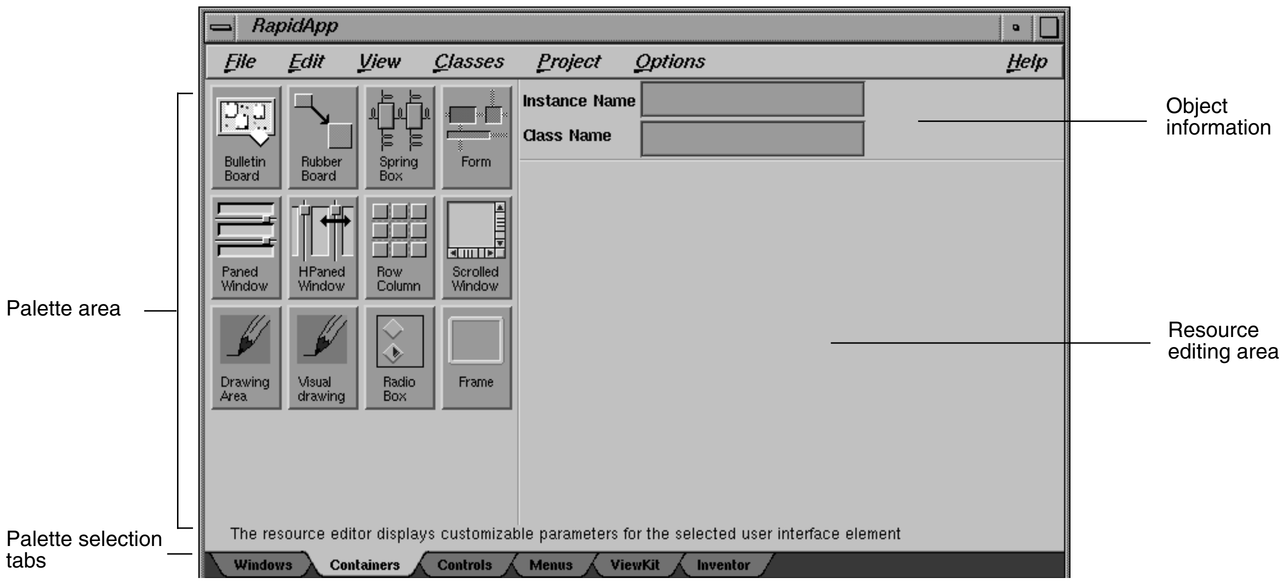
Table 9 Where to Find IRIX IM Analyzer information in the *Developer Magic: IRIX IM Analyzer User's Guide*

Topic	**See ...**
General information	Chapter 1, "Getting Started With the IRIX IM Analyzer"
IRIX IM Analyzer tutorial	Chapter 2, "Using the IRIX IM Analyzer: A Sample Session"
General reference information	Chapter 3, "IRIX IM Analyzer Reference"
Setting breakpoints to capture widget-level information	"Breakpoints Examiner"
Tracing widget-level data through the execution of a program	"Trace Examiner"
Getting information on a specified widget	"Widget Examiner"
Displaying a graph of the widget hierarchy	"Tree Examiner"
Getting information on a specified callback	"Callback Examiner"
Getting information on a specified window	"Window Examiner"
Getting information on a specified X event	"Event Examiner"
Getting information on a specified graphics context	"Graphics Context Examiner"
Getting information on a specified pixmap	"Pixmap Examiner"

Building Application Interfaces With RapidApp

RapidApp is a simple, interactive tool for creating application interfaces. It's integrated with the other WorkShop tools to provide a complete environment for developing object-oriented applications quickly and easily. RapidApp generates C++ code, with interface classes based on the IRIS ViewKit toolkit and IRIX IM. RapidApp also includes predefined interface components that allow you to conveniently use other Developer Magic libraries such as OpenGL™ and Open Inventor™. Applications produced by RapidApp are automatically integrated into the Indigo Magic Desktop environment, letting you take advantage of Silicon Graphics' interface and desktop technology.

Working with RapidApp is similar to using a drawing tool such as Showcase™. A typical RapidApp window is shown in Figure 17. RapidApp lets you create interface elements by clicking icons representing widgets or components in the palette area, positioning them in a template window, and setting their resources in the editing area.



The resource editor displays customizable parameters for the selected user interface element

Figure 17 RapidApp Window Displaying Container Palette

RapidApp User Model

RapidApp users should be familiar with IRIX IM, IRIS ViewKit, and C++ programming. Here's the basic user model:

1. Invoke RapidApp by typing **`rapidapp`** in the directory in which you wish to build your application.

   The RapidApp window is displayed as shown in Figure 17. There are six palettes of icon widgets available. The number of palettes and icons available will increase over time as new, useful widgets are developed. The palettes and icons are:

   - Container palette—provides container widgets, that is, widgets that can hold other widgets
   - Controls palette—provides miscellaneous widgets, typically for controls, fields, and so on.
   - Windows palette—provides simple or special-purpose windows and window-oriented controls, such as menu bars and pulldown menus
   - ViewKit palette—provides ViewKit components, that is, prepackaged assemblies of widgets from the ViewKit libraries
   - Inventor palette—provides viewers, editors, and drawing areas compatible with IRIS Inventor™

   The process is then one of selecting containers, populating them with widgets, and assembling them into elements of your user interface.

2. Select a container widget.

   A rubber-band box appears, representing the initial default size of the widget. Use the mouse to drag it to a working area on your desktop (or inside another container). After you've positioned the new container widget, you can adjust its size by dragging the corners.

3. Edit the widget's resources.

   Customize the widget for your application. RapidApp changes the resource editing area according to the type of widget you are working with. It displays text fields for string resources, radio buttons for Booleans, and menus for resources with multiple values. Figure 18 illustrates the creation of a drawing area container widget. The drawing area icon has been selected from the container palette, the new widget has been placed, and the resource editing area has changed accordingly.

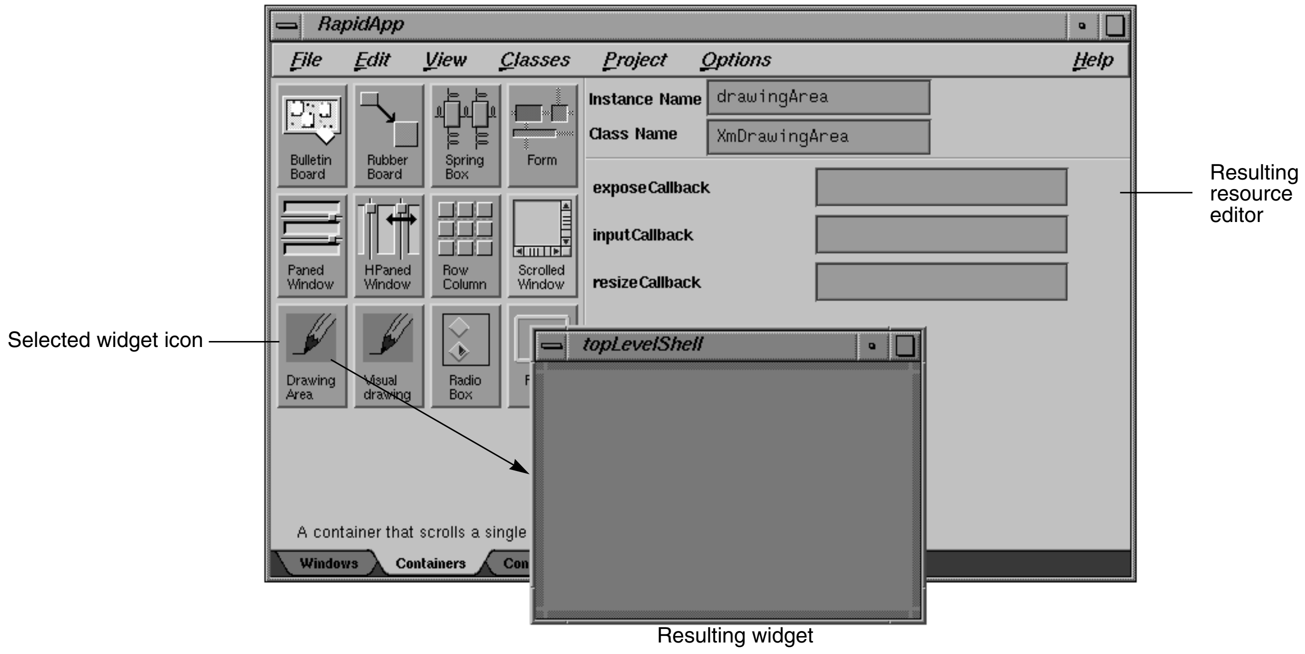


Figure 18 Creating a Widget

4. Select “Play Mode” from the View menu.

   This lets you try out the interface design. When you are through trying it out, go back to working on the interface by selecting “Build Mode” from the View menu.

5. Perform any further edits on the widget.
6. Repeat steps 2-5 until your window (or application) is complete.
7. Select “Generate C++” from the Project menu.

   This produces the source code (including Makefile) necessary to implement the interface you have designed. It also displays the Builder information window, a shell that displays RapidApp status messages.

8. Select "Edit File ..." from the Project menu to make any necessary adjustments to the source code.

   A file selection dialog box displays showing the contents of the directory containing the generated source files. When you choose a file, it will appear in your default editor.

9. Select "Build Application" from the Project menu to compile the new program.

   The WorkShop Build View displays and starts a compile going and lets you view any compile errors (see "Recompiling Within the ProDev WorkShop Environment With Build Manager" on page 30).

10. Use the other ProDev WorkShop and MegaDev tools, if necessary, to fix any coding problems.

    RapidApp is fully integrated with the rest of the Developer Magic environment so that the full range of tools and libraries are at your disposal for completing your application.

Where to Find RapidApp Information

To find out more information about RapidApp, refer to Table 10.

Table 10 Where to Find RapidApp Information in the *Developer Magic: RapidApp User's Guide*

Topic	See ...
Understanding the RapidApp window	"The RapidApp Interface" in Chapter 1
Using RapidApp	"Basic Interaction Techniques" in Chapter 1 and "Advanced Techniques and Features" in Chapter 3
General tutorial	"A Simple Example: A Calculator" in Chapter 1
Inventor tutorial	"Example: A Simple Inventor Program" in Chapter 4
OpenGl tutorial	"Example: Using the OpenGL Widget" in Chapter 4

Table 10 **(continued)** Where to Find RapidApp Information in the *Developer Magic: RapidApp User's Guide*

Topic	See ...
Windows	"Choosing and Using Windows" in Chapter 3
Containers	"Choosing and Using Containers" in Chapter 3
Generating software code	Chapter 2, "Creating Applications with RapidApp"
Applying the other ProDev tools to RapidApp applications	"Integration with WorkShop for Building and Debugging" in Chapter 2
RapidApp reference information	Appendix C, "Quick Reference"

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics to comment on:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please include the title and part number of the document you are commenting on. The part number for this document is 007-2582-001.

Thank you!

Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.

If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs

You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964