SPARCworks 3.0 Overview

SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 800-6943-10
Revision A, January 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc. and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Sun Microsystems Computer Corporation, the Sun Microsystems Computer Corporation logo, SunSoft, the SunSoft logo, SunPro, SunPro logo, ProWorks, ProWorks/TeamWare, ProCompiler, Sun-4, SunOS, Solaris, ONC, ONC+, NFS, OpenWindows, DeskSet, ToolTalk, AnswerBook, and Magnify Help are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc, a wholly owned subsidiary of Novell, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Contents

SPARCworks 3.0 Overview: SPARCworks Toolset

An Introduction to the Toolset

SPARCworks™ is an integrated set of six programming tools for use with SPARCompiler™ language systems. The SPARCworks programming environment speeds and simplifies the tasks software developers do most often — edit and merge source code, compile programs, debug programs, and tune program performance.

SPARCworks is included in the SunPro WorkShop™ for C, C++, FORTRAN, and in SPARCworks Professional Pascal. SunPro WorkShop products are integrated product suites for individuals and teams of software developers that support the rapid development of single and multithreaded applications. The most comprehensive package available, the SunPro WorkShop integrates the tools used by developers on a daily basis: browsers, debuggers, performance analyzers, compilers, and code management tools into a single, highly functional, cost effective package.

The SPARCworks programming environment consists of the following six tools

- **SPARCworks Manager** — for managing and coordinating other SPARCworks programming tools
- **SourceBrowser** — for statically analyzing source code and program structure
- **Debugger** — for dynamically analyzing programs, observing behavior while the program runs, and collecting information for performance profiling
- **Analyzer** — for tuning program performance, including memory allocation
- **FileMerge** — for merging source files and coordinating source code changes
- **MakeTool** — for building programs and browsing makefiles

Compatible Compilers

The following SPARCompiler language systems can be used with SPARCworks tools:

- SPARCompiler C
- SPARCompiler C++
- SPARCompiler FORTRAN
- SPARCompiler Pascal

SPARCworks/Ada is a separate toolset for use with SPARCompiler Ada.

SPARCworks Manager

SPARCworks Manager provides an easy-to-use, unified way to start SPARCworks tools and control the programming environment. Figure 1 shows the SPARCworks Manager window with SPARCworks tool icons displayed. Each tool can be started by double-clicking on its icon or by dragging the icon onto the screen workspace. A tool that is started from the SPARCworks Manager is thereafter associated with that instance of SPARCworks Manager, and can be controlled as part of a management *session*.

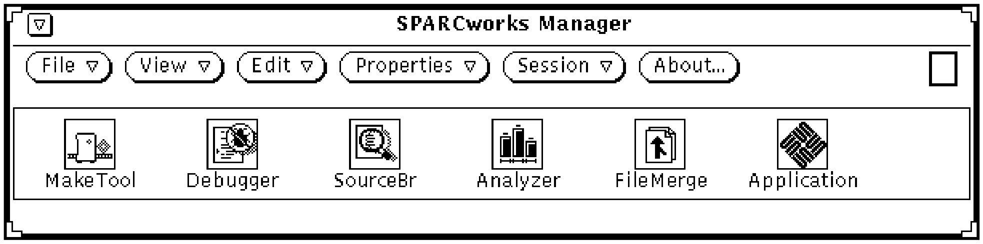
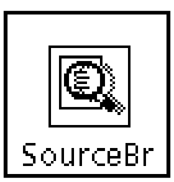


Figure 1 SPARCworks Manager Window

In addition to managing the other five standard SPARCworks tools, SPARCworks Manager can also control custom applications written by users or third-party developers. When integrated with SPARCworks and executed under SPARCworks Manager, the standard SPARCworks tools and custom applications combine to provide you with the following advantages:

- **Session control** — You can group development activities into convenient management *sessions*. Multiple sessions can proceed simultaneously, each controlled by its own SPARCworks Manager.
- **Workspace organization** — SPARCworks Manager helps you make the best use of on-screen workspace. SPARCworks tools associated with a management session can be closed to a single icon.
- **Central control** — The SPARCworks Manager can set environment variables and the working directory for all SPARCworks tools used in a session.

SourceBrowser



SourceBrowser is the SPARCworks tool for static program analysis. Static analysis of a program involves examination of its code. The program does not need to be executing during static analysis. Searching through-level source code with a text editor is a primitive form of static analysis.

SourceBrowser can perform “queries” on source code (analogous to searching in a text editor) that are constrained by the semantics of the source language and by other, user-imposed constraints. In addition, SourceBrowser maintains a list of previous queries so that it can return to view earlier results, eliminating the performance penalty that would be required to repeat the query.

When SPARCompiler language systems execute with the `-sb` command option, they create a database that SourceBrowser uses for searches, allowing it to search large, multifile applications. SPARCompiler language systems also inform SourceBrowser of exactly which source files were compiled to produce an executable or library so that they can be loaded and browsed automatically. By default, SourceBrowser queries all source files, but you can limit the query to focus on any subset of source files. SourceBrowser considers all header files to be source files during queries.

Unlike the SPARCworks Debugger, which uses symbols embedded in the compiled executable, SourceBrowser uses its own database of symbols that is independent of the executable. This arrangement has two benefits: it speeds SourceBrowser search activities and minimizes the amount of information that the compiler must include in executables to enable SourceBrowser to do its work.

In situations where you do not have a database, you can still perform grep queries with the SourceBrowser, which provides a graphical and buffered access to grep. All of the standard SourceBrowser features for browsing a query apply to a grep query.

SourceBrowser Window

SourceBrowser operates by responding to user *queries* that instruct it to find all occurrences of a specified identifier, string constant, or search pattern.

Figure 2 shows the SourceBrowser base window. The window contains a control area filled with buttons, button menus, and a field where you enter text before instructing SourceBrowser to search for matches. The control area also displays a message that reports on the number of matches that result from a query.

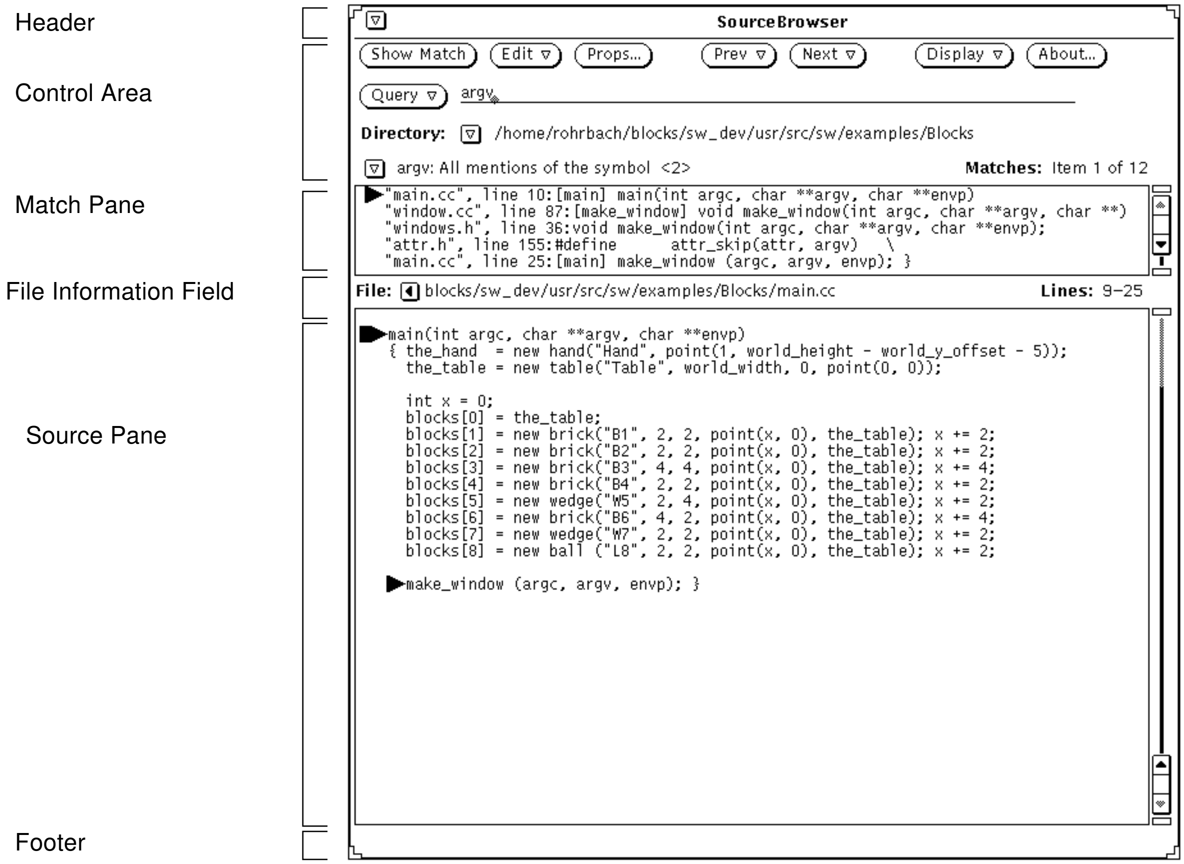


Figure 2 SourceBrowser Base Window

Below the control area is a scrollable match pane that displays all matches found by the current query. Below the match pane is the source pane, which displays the contents of the source file in which a match was made. Text above the source pane lists the current working directory, the name of the file being displayed, and the line numbers of the first and last lines being displayed.

Beneath the source pane is a message area that displays the type and value that is the subject of the current query.

SourceBrowser Capabilities

This section highlights the features and capabilities of SourceBrowser.

Collecting Information Without Compilation

Scanning, a quick and convenient method for collecting browsing information from source files, enables you to collect browsing information about programs that do not compile completely. Scanning is also useful when you want to speed up compilation time by not issuing the `-sb` option, resulting in a smaller database. Scanning is accessed with the `sbtags` command; the `sbtags` command collects a subset of the information available through compilation.

Grep Queries

Even without compiling or scanning your source code, you can use the SourceBrowser as a graphical user interface to `grep`. SourceBrowser executes the grep program and displays the matches found by the `grep` command in the Match panel. The source file containing the current match is displayed in the Source Window. You can view different matches by using the Prev and Next menus. Grep queries are saved together with other queries, and you can go back to any query by using the query history abbreviated menu button.

Finds Symbols and Strings in Source

In a SourceBrowser session, you issue queries that instruct SourceBrowser to find all occurrences of a specified symbol, string constant, identifier, or search pattern. In response, SourceBrowser displays *matches* (occurrences of the text that SourceBrowser was ordered to find) with your surrounding source code.

SourceBrowser maintains a list of all queries that were conducted during a session. This feature makes it easy to issue a query, view the resulting matches, conduct another query, and then return to the original query with a minimum number of commands.

Matches Wildcards and Regular Expressions

In addition to literal text, query patterns can contain *wildcards*, as used in shell-style patterns, or *regular expressions*, as used in `vi`, `grep`, and other Solaris editors and utilities. The SourceBrowser Properties Window lets you specify which convention to use.

Performs Semantic Queries

SourceBrowser can respond to queries that filter matches according to the semantics of the source language. For example, a developer using the SPARCompiler could issue a very specific query such as "Show all occurrences of the variable *age* when *age* is used as a structure field name." The search can be further focused on specific items of certain classes of code, such as files or functions.

Supports Many Languages

SourceBrowser has no built-in, architectural affinity for any specific language. Instead, SourceBrowser obtains information about each supported language by reading a configuration file. This lets SourceBrowser support programs that mix software modules written in different languages.

Provides Text Editor for Source

You can edit source directly in the text pane. The text pane uses Text Editor commands that are the same as those in other OpenWindows applications such as MailTool. Files under the control of SCCS can be checked out directly from a menu in the text pane.

If the Source Browser is invoked with a `-editor` flag, it does not show the source text window. Instead it uses ToolTalk to show the source in an external editor. If you have a ToolTalk enabled editor, it can be used to display the source.

Handles Large Applications

The first time SourceBrowser runs a query it creates an index file for use in subsequent queries. The index file allows SourceBrowser to maintain a high level of efficiency regardless of the amount of code it is browsing.

SourceBrowser also includes several features specially designed to improve performance while browsing large amounts of code. In one experiment, all of `/usr/src` for the Solaris was compiled for browsing with SourceBrowser. The resulting SourceBrowser database was between 40 and 50 Mbytes. Even with such a large database, SourceBrowser was able to respond to queries in one or two seconds after it had created its index file.

Produces Call and Class Graphs

SourceBrowser can produce function call graphs for procedural languages and class graphs for SPARCompiler C++. Call and class graphs are valuable overviews of function and object relationships, especially for large programs.

Simple call and class graphs are shown in Figure 3. The graphs show relationships among functions and classes for simple programs. Horizontal and vertical scroll bars are provided for viewing larger graphs.

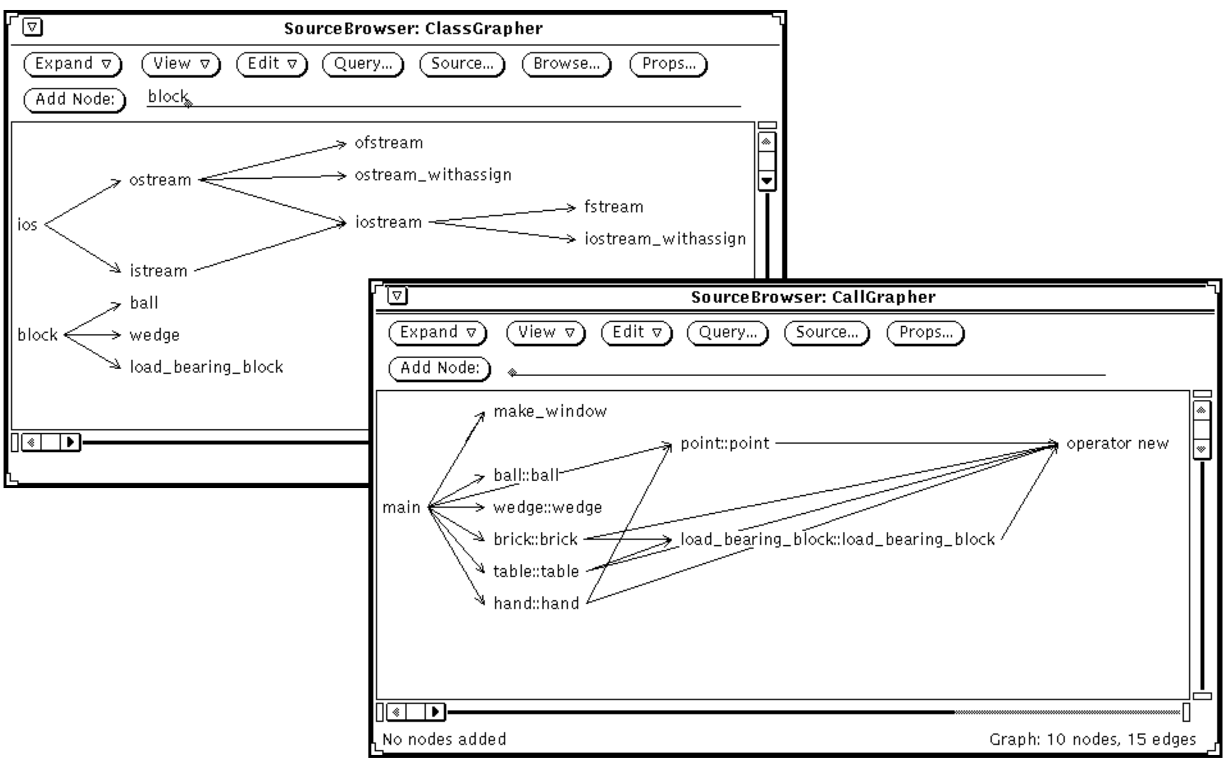


Figure 3 SourceBrowser Call and Class Graphs

Note — If the database was generated with `sbtags`, the SourceBrowser does not have any information for browsing or graphing C++ classes. However, you can perform queries on functions and global variables and you can display function calls.

Class Browser

In addition to the Class Grapher, SourceBrowser provides another tool to help understand how SPARCompiler C++ classes are defined and used. Class Browser, activated from SourceBrowser, helps you effectively browse through C++ source code and libraries. The Class Browser base window is shown in Figure 4.

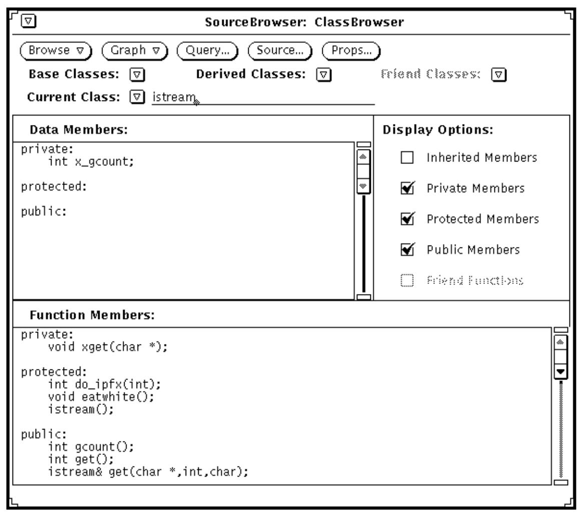


Figure 4   Class Browser Base Window

The Class Browser provides the following capabilities:

- **List and select from available classes** — view an alphabetical list of available classes and select one from the list to display information.
- **Navigate from class to class** — easily follow inheritance links to browse information.
- **Display class data and member functions** — view the data members and member functions of a specified class, including those that are inherited from ancestor classes.
- **Show access protection of class members** — display access restrictions for data members and member functions to reveal valid class-defined interfaces.
- **Identify friend classes** — discover the friend classes and functions of a specified class.
- **Display the source of member functions** — select a function and display its source code.

- **Issue queries for class usage** — use the SourceBrowser query facility to find usages of a specified class.
- **Interact with Class Grapher** — select a node in Class Grapher and display class information in Class Browser.

Using SourceBrowser

You initiate a SourceBrowser query in one of two ways:

- Select a text string anywhere on the screen and click the Query button
- Type a string in the text entry field next to the Query button and press the Return key

The first method makes use of the selection service and is especially useful when SourceBrowser is used with other OpenWindows programs such as the SPARCworks Debugger.

Filtering To Obtain Useful Matches

The number of matches the SourceBrowser can handle is limited only by the available swap space. For a symbol like `NULL`, a query might produce such a large number of matches that the information present is not useful. SourceBrowser filter capabilities can be used to reduce the number of matches.

Language-Specific Filtering

You can instruct SourceBrowser to restrict its search on the basis of semantic information specific to a particular language. The choices on the Filter popup window are language-specific and change automatically with the languages that have been compiled into the SourceBrowser database. For example, you can specify that SourceBrowser search only for C-language declarations of the symbol in the popup window (Figure 5) produced by choosing Filter from the Query menu.

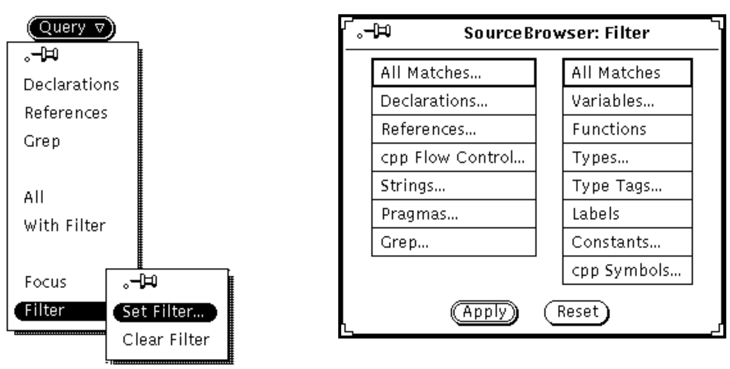


Figure 5 Setting a Filter for C Language Declarations

The choices shown in Figure 6 are for SPARCompiler FORTRAN.

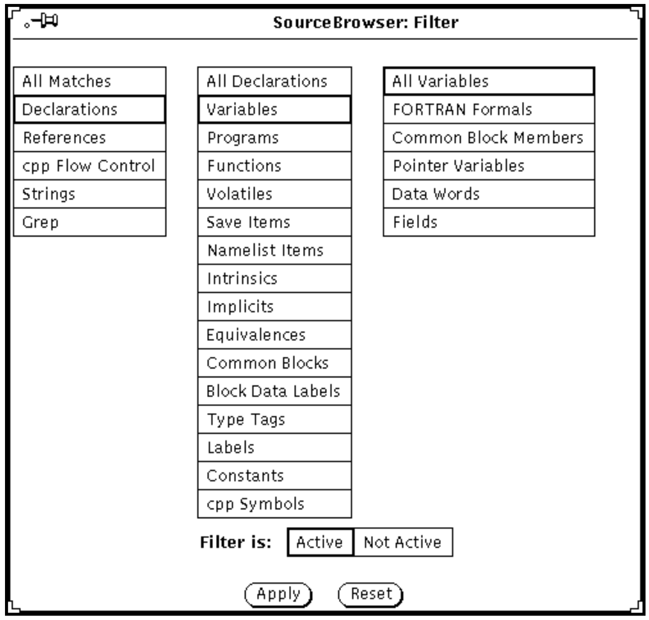


Figure 6 Filter Choices for SPARCompiler FORTRAN

When a project requires that object files be compiled from sources written in different languages, the linker (`ld`) records this fact in the SourceBrowser database at link time. The Filter pop-up window shows the union of all semantic constructs for the languages that were used to produce the final executable.

Focusing

In addition to filtering, you can restrict SourceBrowser to query only certain sections of the source: a specific language, function, or set of source files, for example. This restriction is known as *focusing*.

In Figure 7, SourceBrowser is being instructed to focus its queries on classes.

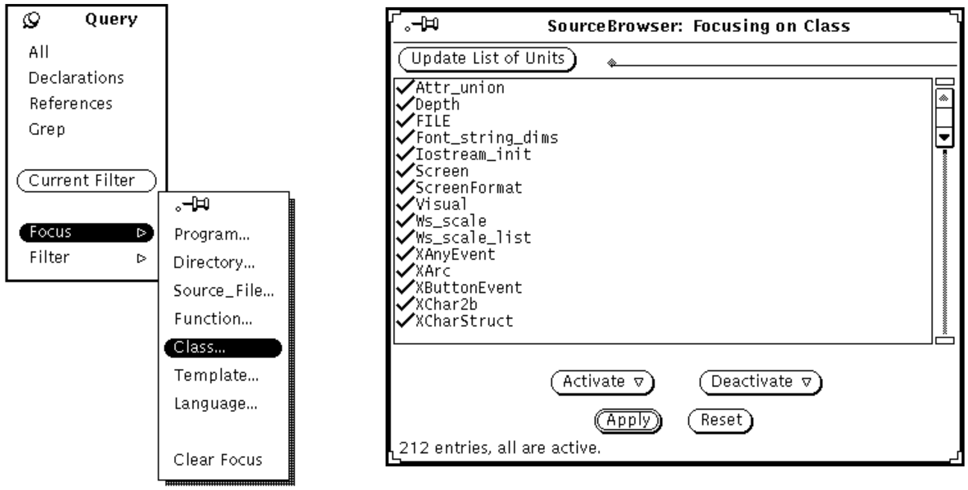


Figure 7 Focusing a Query

Any number of items in the list can be activated; SourceBrowser thereafter limits its queries those parts of the source that have been activated. Matching Strings

The information stored in the SourceBrowser database is determined by the compiler that generates the database—SPARCompiler language systems have been designed to store strings and symbols. Strings are stored along with the quotation marks that are used in the language, which identifies them as strings and enables SourceBrowser to perform queries on them (to limit a query to strings only, set a filter for Strings).

Queries for strings are very useful for debugging an application that has printed a message (typically an error message). You can search for a string in the message and find the place in source where the message is printed. In conjunction with Debugger, you can then determine the exact circumstance that caused the message to be printed and rapidly uncover the source of the error.

Checking Out Files from SCCS

The source pane in SourceBrowser can be enabled as an editing window, with features identical to OpenWindows Text Edit. A popup menu in the source pane enables editing (Figure 8). If the source is controlled by the Source Code Control System (SCCS), the same menu can check out the source file for editing. This feature is also found in the Debugger source window — source files can be checked out from SCCS and edited in either window.

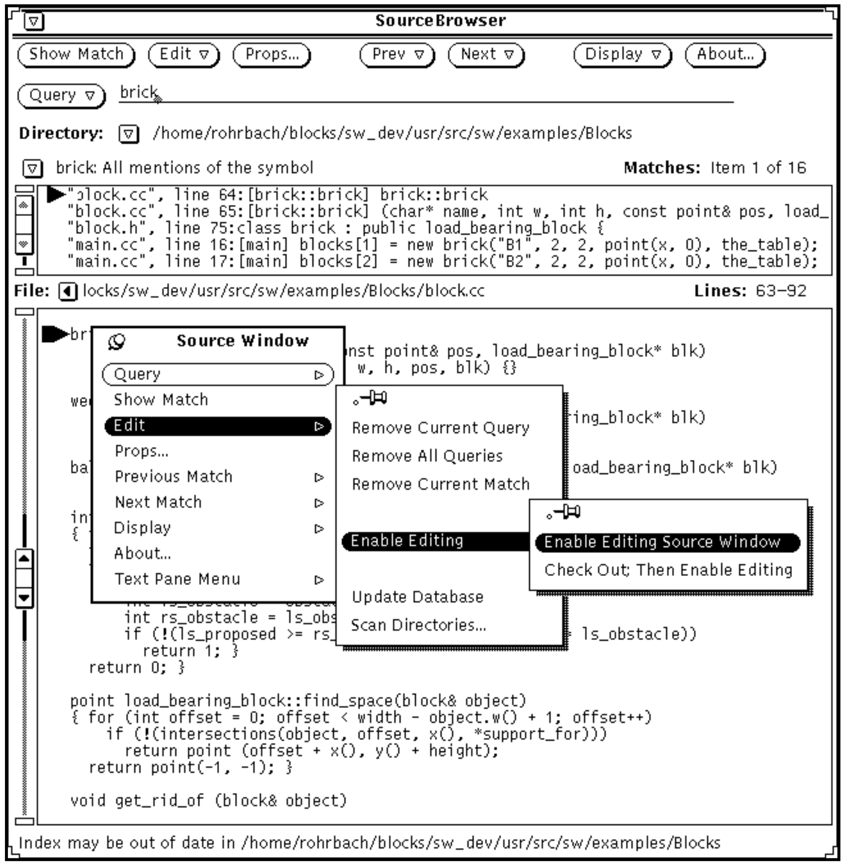
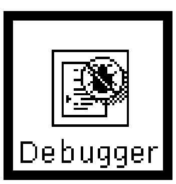


Figure 8 Checking Out a File for Editing in the Source Pane

Debugger



The SPARCworks Debugger dynamically analyzes a program, letting you observe its behavior while it is running. Debugger gives you complete control of program execution and simultaneously collects performance data for later use with SPARCworks Analyzer. From within Debugger you can identify a problem, edit source code, rebuild the program, and then continue dynamic analysis.

The Debugger has many advanced features, such as integrated RTC to improve quality, Fix and Continue to improve productivity, Visual Data Inspector to understand complex data structures, and Process Inspector to assist in the development of threaded applications.

In addition to the mouse-selectable functions of the OpenWindows interface (which you can customize to suit your needs), Debugger provides a pane in which you can type commands directly to the command line.

Debugger Features

The following list summarizes the activities you can perform with Debugger and gives an overview of its many features.

General Features

- **Display source code** — the Debugger displays source code corresponding to the machine code being executed.
- **Analyze multiple languages** — Debugger supports SPARCompiler C, C++, FORTRAN, and Pascal language systems.
- **Analyze low-level code** — the Debugger analyzes assembly language code as well as code written in high-level languages.
- **Analyze live processes** — attach Debugger to an executing program such as a daemon, debug the process, and then detach Debugger. Programs do not need to run as children of Debugger in order for you to analyze them.
- **Analyze dynamically linked shared libraries** that have been loaded at program startup or through the `dlopen()` system call.
- **Analyze optimized code** — code compiled with both the `-O` and `-g` option can be analyzed.
- **Load symbol-table information automatically** — load compiled symbol table information on demand. Makes starting Debugger with a large program much faster.
- **Save, restore, and replay sequences of commands** — save analysis sessions in the form of command sequences. The sequence can then be replayed to restore the analysis session, or a subset of the sequence can be replayed for an "undo" effect (on programs that are determinate).

Control Program Execution

- **Single step** through the program by machine instructions or by source lines. You can decide to step into procedures or over them.
- **Set breakpoints** at source line, address, exception, instruction count, procedure, or function. You can also set conditional breakpoints.
- **Set watchpoints** for variables, functions, source lines, or expressions you enter at the command line. Watchpoints are a class of events that provide the Debugger the capability to note when the value of a variable or expression has changed and fired an event. You can execute a command or evaluate an expression when the value of the watchpoint changes. You can also set conditional watchpoints and watchpoints that focus on a particular function or procedure.
- **Set Postbreak Modifiers** for breakpoints and tracepoints. A postbreak modifier instructs Debugger to test for a condition after the program arrives at a tracepoint or stops at a breakpoint.
- **Manage events** with the Debugger's custom handlers to create and manipulate event handlers so that when specific events in the program occur, certain operations are performed in the program being debugged.

- **Handle signals** — intercept signals and act on them; ignore other signals that might halt execution.
- **Step out of functions with a single command** — after you have stepped into a function, a single command lets you quickly return to the calling function.

Evaluate Functions and Monitor Data

- **Examine program variables** including complex structures and monitor values during program execution with the Visual Data Inspector (VDI). You can also inspect two-dimensional arrays in a spreadsheet-like format.
- **Modify data** — edit code at source level; change the values of variables at a low level.
- **Monitor the value of expressions and variables** — the Data Display popup window allows you to view the values of program variables, including the values of nested pointers.
- **Set tracepoints** for variables, lines, and expressions in lines. A tracepoint is the location in a program where a trace begins. You can set bounded traces for all statements in functions, member functions, classes, or entire programs. You can specify `trace` subcommands, such as echoing the identified code as it executes or printing the values of variables or arguments passed to functions, or the names of member functions.
- **Navigate the call stack** — move up and down the call stack.
- **Inspect the call stack** with the Stack Inspector to view the call stack, to hide or show selected functions, and to move from one frame to another.
- **Handle overloaded names** — when variables from different functions or procedures have the same name, you can differentiate each name by fully qualifying it with the name of the function (and source file, if necessary) in which it occurs.
- **Evaluate operators and native language expressions**.

Language Specific Functionality

- **Catch exceptions** — the Debugger can also help you catch exceptions. While debugging, you can
  - catch unhandled exceptions before stack unwinding
  - catch specific exceptions whether handled or not before stack unwinding
  - determine where a specific throw would be caught if it occurred a at particular point in the program
- **Handle overloaded C++ function names** by popping up the Overload Display window when you search for a C++ member function with an ambiguous name or an overloaded function name. You can then select the function you want to visit from a list of functions with the overloaded name in the window. Note that if a function is called with arguments or the function's prototype is used, the Debugger will automatically determine which function to call.
- **Call functions with default arguments** — the Debugger encodes unnamed arguments in a form that allows you to evaluate.

- **Evaluate array slices** — the same way you evaluate other types of variables: select the array, then choose Evaluate from the Data menu. You can evaluate part of a large Fortran array.

Advanced Features

- **Check for runtime errors** — the Debugger's Runtime Checking (RTC) feature lets you automatically detect runtime errors during the development phase. Two kinds of runtime errors RTC checks for are memory access errors and memory leak errors.

  RTC detects the following types of memory access errors:

  - Read from uninitialized memory (rui)
  - Read from unallocated memory (rua)
  - Write to unallocated memory (wua)
  - Write to read-only memory (wro)
  - Misaligned read (mar)
  - Misaligned write (maw)
  - Duplicate free (duf)
  - Bad free (baf)
  - Misaligned free (maf)
  - Out of memory (oom)

  RTC also prints the following information for memory access errors:

type	type of error.
access	type of access attempted (read or write).
size	size of attempted access.
addr	address of attempted access.
detail	more detailed information about addr. For example, if addr is in the vicinity of the stack, then its position relative to the current stack pointer is given. If addr is in the heap, then the address, size, and relative position of the nearest heap block is given.
stack	Call stack at time of error.
location	where the error occurred. If line number information is available, this information includes *filename*, *line number*, and *function*. If line numbers are not available, RTC provides *function* and *address*.

  RTC detects the following memory leak errors:

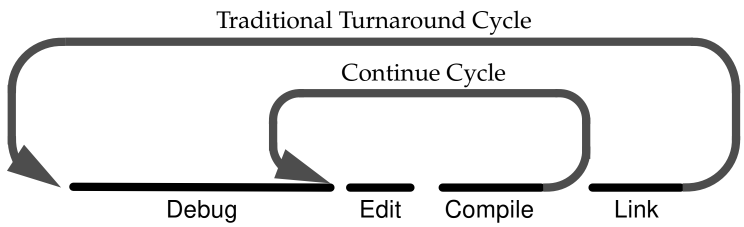
  - Memory Leak (mel)
  - Possible leak -- Address in Register (air)
  - Possible leak -- Address in Block (aib)

  RTC prints the following information for individual memory leak errors:

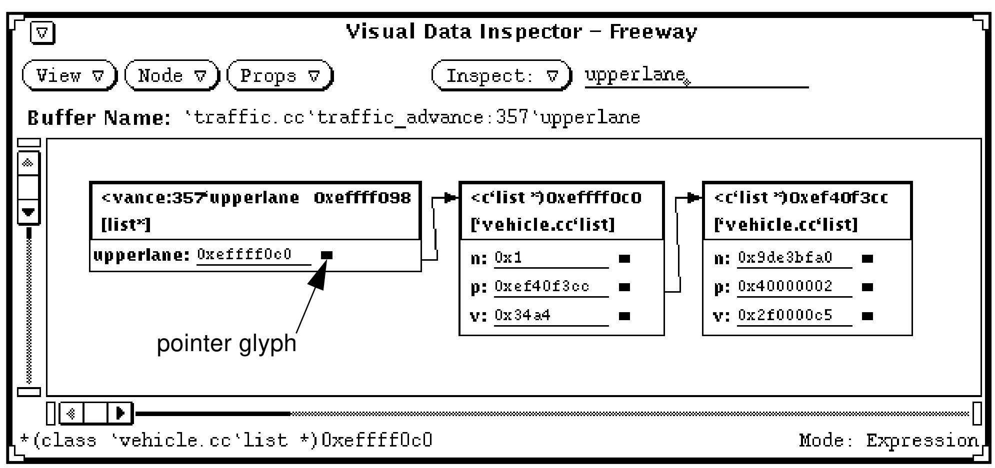
location	location where leaked block was allocated.

addr	address of leaked block.
size	size of leaked block.
stack	at least *stackdepth* preceding functions in the call stack at time of allocation.

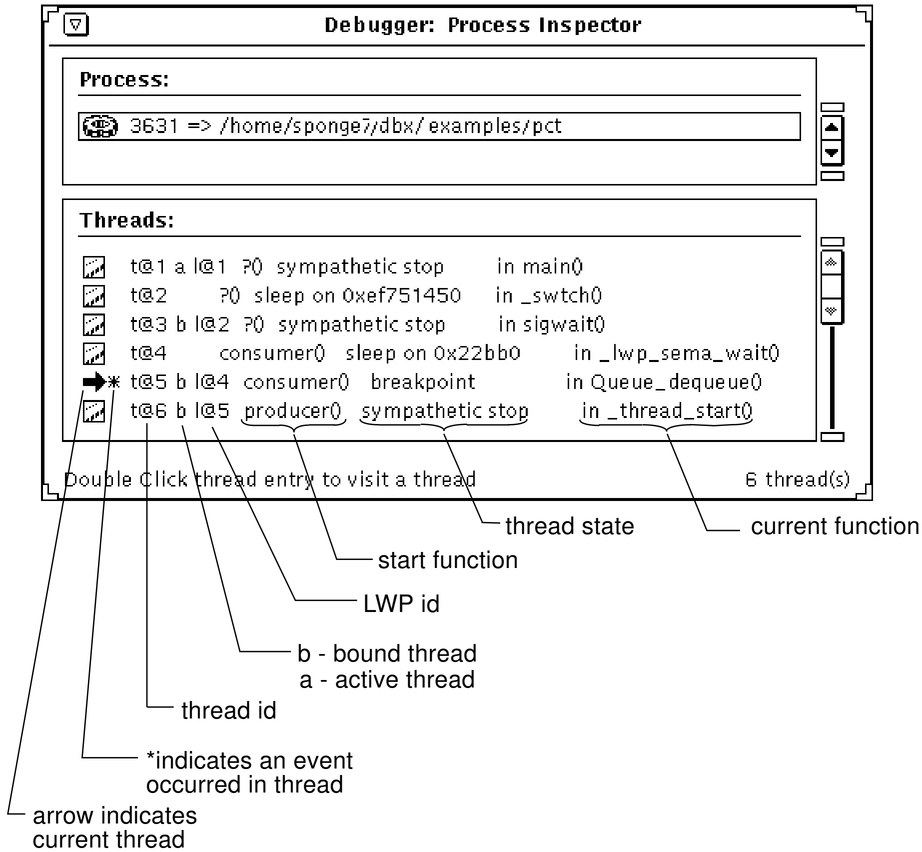
- **Fix source code and continue program execution** — use the `fix` command to modify source code without having to leave the Debugger. The modified file is recompiled, and the running process is returned. You can continue the program from the point where it stopped.



- **Follow forking process** — if a child process executes `exec(2)`, the Debugger can either detach from the parent process and attach to the child, or start a new Debugger engine to debug the child process.
- **View data with the Visual Data Inspector (VDI)** — use the VDI to track and graphically display variables, including data structures. You can minimize or expand the display of information and follow pointers to other variables.



- **Inspect threads** with the Process/Thread Inspector. The Debugger recognizes a multithreaded program and automatically enables its multithreaded features. You can also find information about light weight processes with the inspector.



Additional Features

- **Call functions** when a program is stopped with the *`call`* command. The command accepts values for the parameters that must be passed to the called function and runs the function. The Debugger honors any breakpoints that you have set in the function.
- **List command history** — a history command provides most of the same functionality as the C-shell `history` command.
- **Assign values to variables** — when used with the `cont at` (continue at) command, this feature enables you to assign known values to variables, skip over faulty code, and continue program execution.
- **Check out source from SCCS** — like the source pane in SourceBrowser, the source pane in Debugger can be enabled as an editing window. If the source file is controlled by SCCS, Debugger can check out the file.
- **Run subprograms**.
- **Extensible** — the Debugger command language is based on the syntax of the Korn shell, including I/O redirection, loops, built-in arithmetic, history, and command-line editing (only in command-line mode, not available from the Debugger).

Debugger Interface

Figure 9 shows the Debugger base window containing the following elements:

- **Button menu control panel** — contains button menus that provide access to the most important and commonly used debugging commands, as well as an assortment of debugging utility commands and property window controls.

  Beneath the buttons, the window displays current program location (by file and function) and the full path name of the source code file being displayed.

- **Source display** — usually displays the current locus of execution in the source code, indicated by a bold arrow to the left of the source line. An outlined arrow indicates an active call on the stack. Stop sign glyphs to the left of source lines indicate where breakpoints are set. When the program is executing in single steps, the source text view in the pane is updated to keep up with the changing locus of execution.
- **Command button control area** — buttons in this area provide easy access to the most commonly used items from the button menus at the top of the window. Users can customize this area by adding buttons to execute any `dbx` command.
- **Command pane** — you can enter `dbx` commands from the keyboard in this pane. The pane echoes the commands generated by control panel buttons. Messages returned by `dbx` also appear in this window.

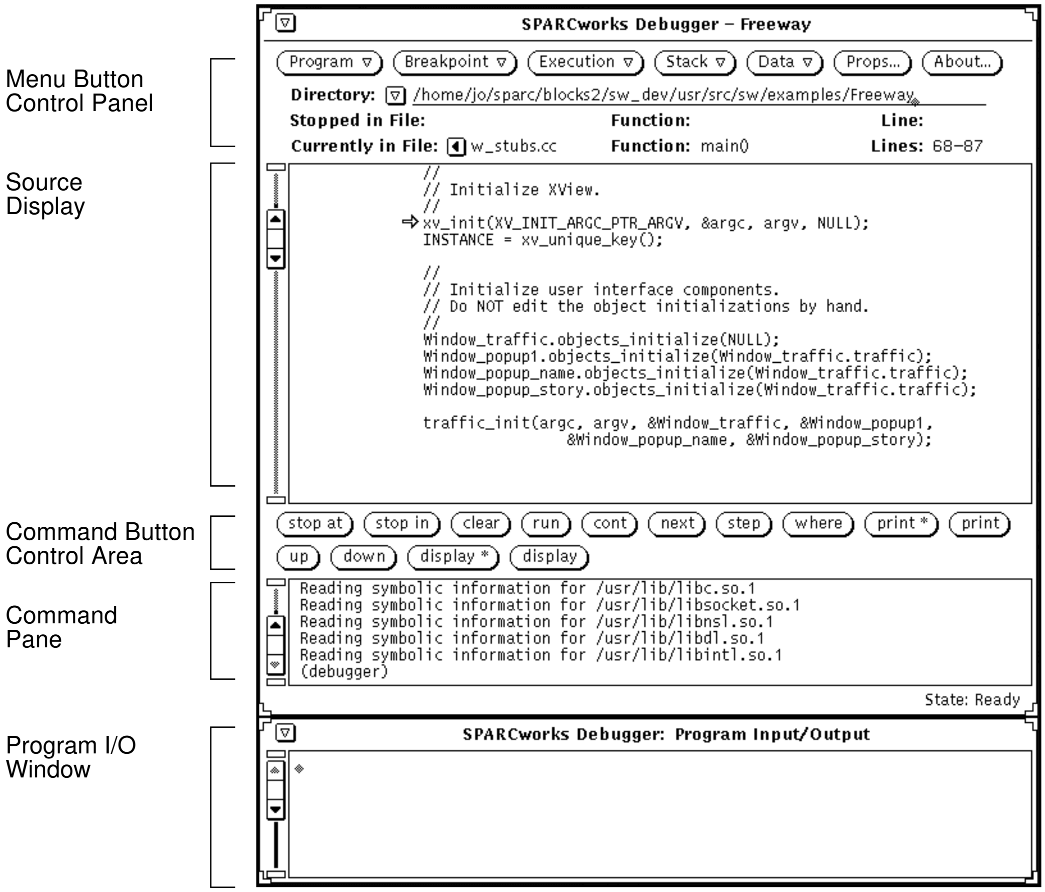


Figure 9 SPARCworks Debugger Window

Collector

In addition to its duties as a dynamic analyzer, the Debugger also collects data for use with the Analyzer performance-tuning tool. The Debugger Collector window sets data collection parameters. As the application executes under the Debugger, the Collector gathers performance and test coverage data and writes the data to a file. The execution run is called an *experiment*; the file is referred to as an *experiment record*. Figure 10 shows the default settings of the Collector.

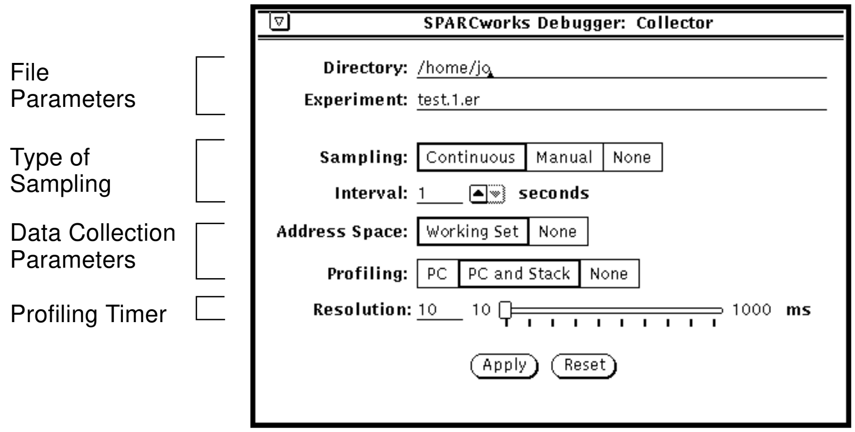


Figure 10 Collector Window

The Collector window is divided into four main areas:

- **File parameters** — in this area you name the directory and file name for the experiment record.
- **Type of sampling** — in this area you choose the type of sampling for the experiment. Each sample contains information on the target program, collected over a specified period of time. Continuous sampling is useful for initially identifying problem areas. As you focus in on a problem, you may want to adjust the sampling interval or trigger manual sampling at a specific point in the program as it executes under control of the Debugger.
- **Data collection parameters** — in this area you specify the types of data to collect during the experiment.

  The Collector always accumulates data for an overview display of the experiment. Additional data collection parameters are:
  - **Working Set** — causes the Collector to accumulate memory use data.
  - **PC** — causes the Collector to accumulate program counter profiling data. This data identifies the functions in which the program spends the most time.
  - **PC and Stack** — causes the Collector to accumulate return addresses on the call stack. This data is useful for profiling applications that have a hierarchical, modular design and for complex applications that make heavy use of standard library files.
- **Profiling timer** — specifies how often data is to be collected.

Analysis of collected data is described in the section “Analyzer” on page 21.

Using SourceBrowser with Debugger

Because SourceBrowser and Debugger use the selection service, they are integrated with each other and with other OpenWindows applications.

When you run SourceBrowser along with Debugger you combine the benefits of static and dynamic analysis. For example:

- When you stop at a function in Debugger, you can then go to the SourceBrowser to find how the function is defined. You can also find all the other places where it is used.
- You can use SourceBrowser to search for a variable. If the search finds the variable in an interesting line of source code, you can move to Debugger, set a breakpoint, and examine the values taken on by the variable in the statement during execution.

A Brief Example

In this example, you use the SourceBrowser in conjunction with the Debugger to search for and set a breakpoint at a specific variable. Before you can synchronize the SourceBrowser and the Debugger, you must start both of them from the same SPARCworks Manager.

To synchronize the tools, follow these steps:

1. **Load a program into the SourceBrowser.**
2. **Use the SourceBrowser to display the line of code where you want to place a breakpoint.**
3. **Ensure that the program you are browsing with the SourceBrowser is properly loaded into the Debugger.**
   Use the Load item in the Debugger Program menu to load your program if necessary.
4. **Set SrcDispSynchronization to Always in the Debugger.**
   In the Debugger main window, click on the Props button to bring up the Properties window and choose Miscellaneous from the Category menu. The SrcDispSynchronization item has two states: Never and Always.
5. **Choose Sync Debugger from the SourceBrowser's Display menu.**
   The code shown in the SourceBrowser now also appears in the Debugger.
6. **Use the Debugger to set the breakpoint in its own source display.**

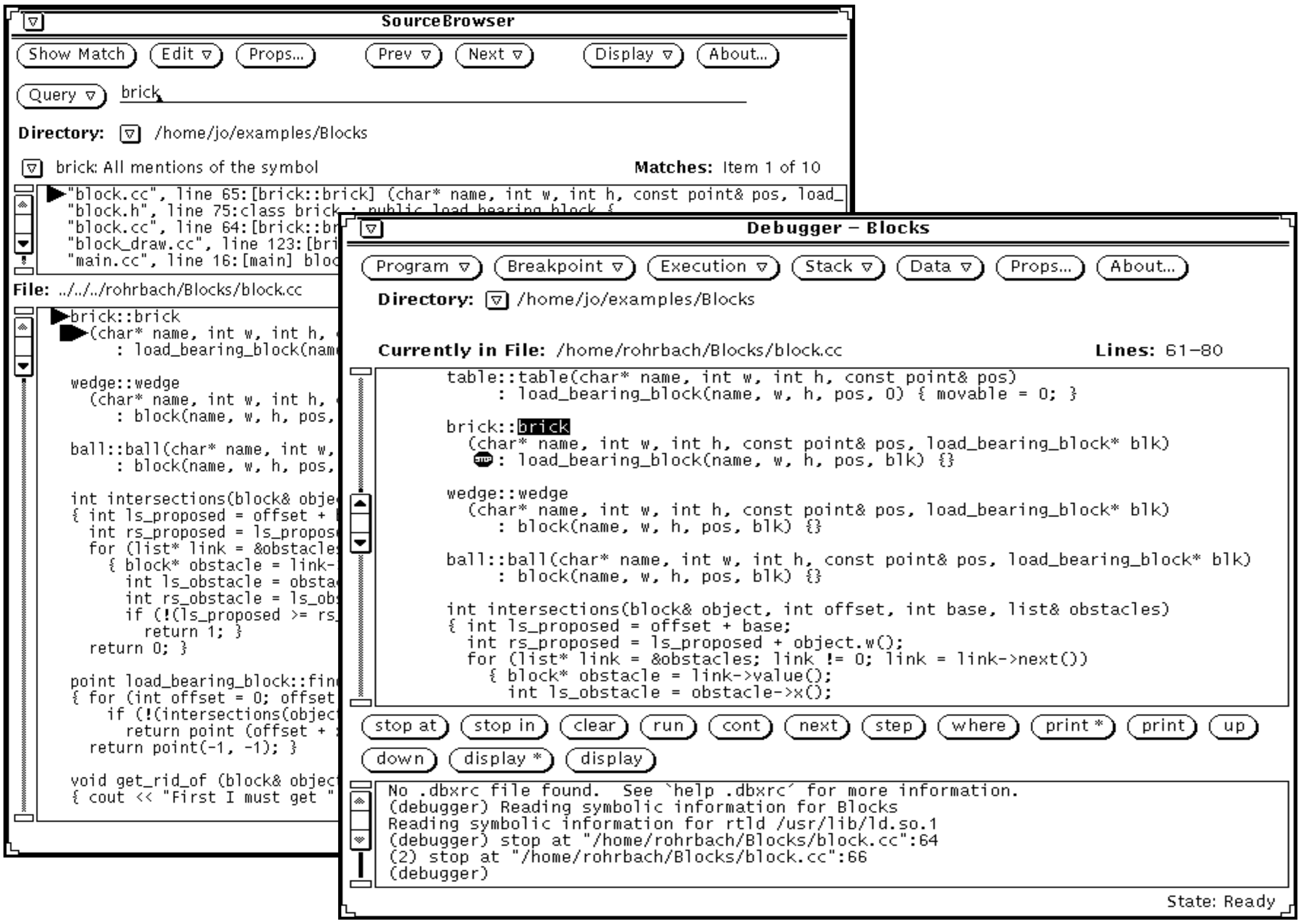
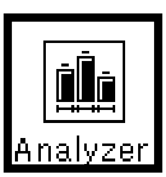


Figure 11 Querying for the Selected Variable in SourceBrowser

Analyzer



After you have successfully compiled a program and eliminated its major bugs, you will want to evaluate its performance. The SPARCworks Analyzer measures and displays an application's performance profile, suggesting ways to improve performance.

Any program is capable of generating data for the Analyzer. The Debugger collects performance data while the program runs under Debugger control. The performance data is placed in a set of files that the Analyzer subsequently examines and presents in a variety of graphic and text displays.

Features

The Analyzer eliminates the need to compile and link an application with special data collection instrumentation — any program can be analyzed. Moreover, the Collector simplifies and enhances the task of collecting data: you can collect a variety of performance data types, and you can control the data collection process while an application is running.

By letting you focus on the areas where performance problems occur, the Analyzer easily tests your hypotheses about a program's behavior.

After you have tuned your program, the Analyzer assists you in rebuilding it. As a further help in rebuilding your program with improved performance, the Analyzer identifies improved ordering for loading functions into the program's address space. It creates a mapfile and the user then rebuilds the program by passing the new ordering to the SunOS linker and produces an executable with reduced working set size.

Collecting Experiment Data

Experiment data is collected by the Debugger, as described in the section titled "Collector" on page 18. Following collection, the data is written to a set of files called the *experiment record*.

Analyzing Experiment Data

The Analyzer examines the performance data that has been collected in the experiment record and displays its results in a variety of graphic and text formats.

Analyzer Features and Displays

The performance data that you can examine in the Analyzer includes

- **User Time** — the amount of time spent executing program instructions.
- **Fault Time** — the time required to service page faults classified into text and data page faults.
- **I/O Time** — time the operating system spent waiting on I/O (input/output) operations, such as writing to a disk or tape.
- **System Time** — the time the operating system spent executing system calls.
- **Trap Time** — time spent in executing traps (except system calls or page faults).
- **Lock Wait Time** — time spent waiting for lightweight process locks.
- **Sleep Time** — time the program spent sleeping (waiting for system resources).
- **Suspend Time** — time spent suspended (includes time spent in the Debugger during breakpoints) and time to write out experiment data.
- **Idle Time** — time spent waiting to run while system was busy.
- **Function Sizes** — the sizes of functions in the program.
- **Module Sizes** — the sizes of modules in the program.
- **Segment Sizes** — the sizes of segments in the program.
- **Memory Usage** — memory page reference and modification data.

- **Resource Usage** — information about the system resources that are used by the program, including major and minor page faults, process swaps, number of input and output blocks, number of messages sent and received, number of signals handled, number of voluntary and involuntary context switches, number of system calls, number of characters of I/O, and number of working set memory pages.

The Analyzer provides four displays in which to view and organize the performance data: Overview, Histogram, Pages, and Statistics.

Overview Display

Figure 12 shows the Overview display, which is automatically displayed after an experiment is loaded. Data for the Overview display is always accumulated when the Debugger Collector has been activated, so the Overview display is always available in the Analyzer. The Overview display is useful for conducting the initial examination of program performance. Each column in the display represents a single sample of experiment data. The columns can be displayed in uniform widths or in widths proportional to the duration of each sample.

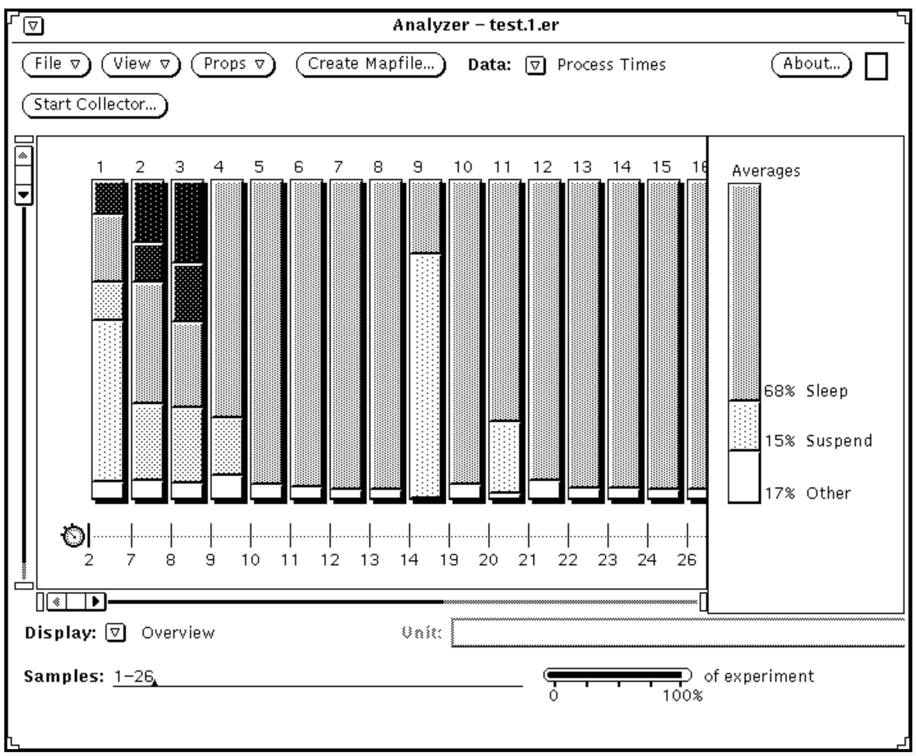


Figure 12 Overview Display

The Overview display can show, in a pop-up window, more detailed performance data specific to a sample you have selected (see Figure 13).

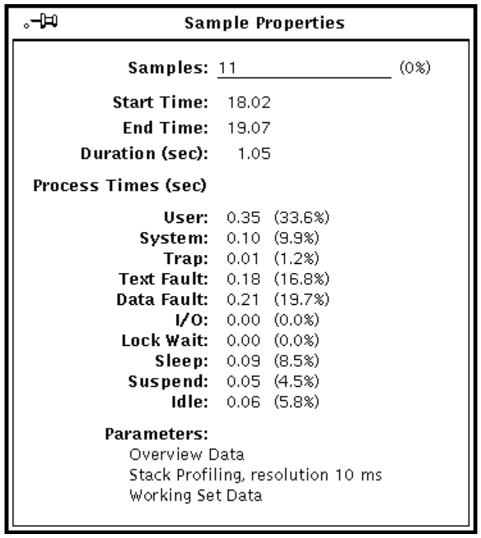


Figure 13 Selection Properties Pop-Up Window

Histogram Display

The Histogram display provides another overall view of application performance, and is considered by many users to be the most useful Analyzer display. The histogram quickly reveals the functions in which the application is spending most user time. After viewing the data in the Analyzer, you may decide to tune the program so that it calls those functions less often, or you may want to rewrite the functions themselves so that they execute more quickly.

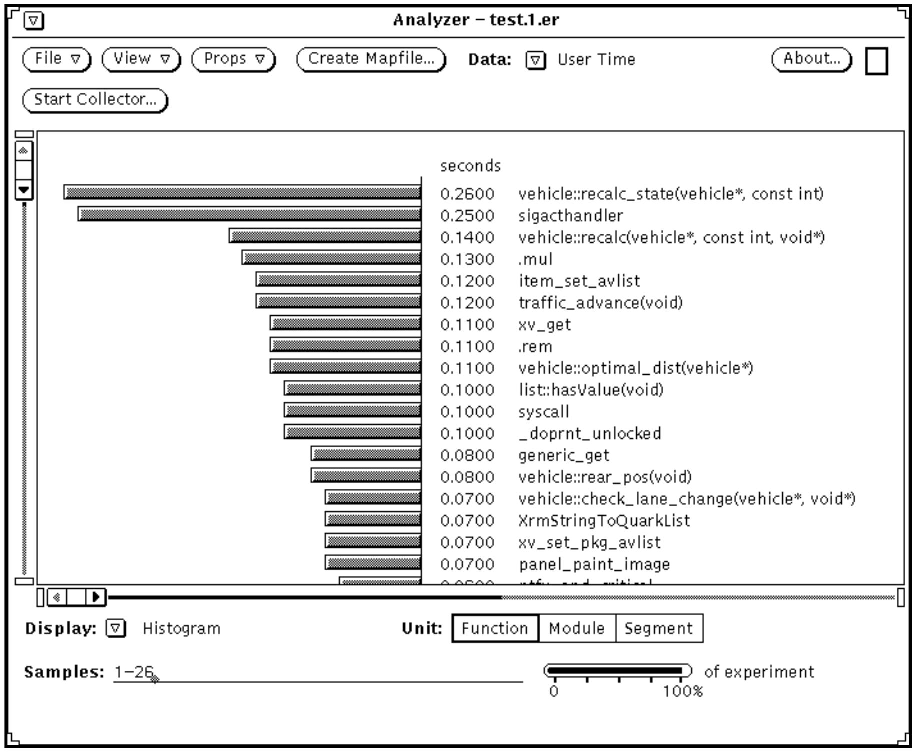


Figure 14 Histogram Display

For display and analysis purposes, the Analyzer dissects the application in one of three ways: according to function, module, or segment:

- **Function** — Function is the default histogram display. This display identifies the functions in which the running program spent most of its time during the experiment. These functions are prime candidates for tuning performance.
- **Module** — The Module histogram display shows a higher level of data aggregation than the function histogram. The module histogram is useful when an application is made up of such a large number of functions that you cannot easily understand performance behavior from viewing a function histogram.

  Modules are units of executable code that correspond to individual source files. This division of the program is sensible because programming practice usually groups related functions into separate source files.
- **Segment** — The Segment histogram display shows yet a higher level of data aggregation. Typical programs are divided into two to eight segments. The program itself is one segment; other segments result when the program dynamically links to shared libraries during an experiment.

The segment level of data aggregation results in a very coarse performance view. The view is useful, however, because you can easily see how much of your application's execution time is spent in the code of a shared library. If a great deal of time is spent in libraries, you may be forced to limit your

performance tuning efforts to calling library functions less often; if little time is spent in libraries, consider improving the performance of your own functions.

You can display histogram data sorted either alphabetically or numerically. To find a specific function, module, or segment, you can use the scroll bars provided and search visually, or use the Find feature to search for text strings in a name.

Pages Display

The Pages display shows the memory usage of process text address space on a memory-page or segment basis (see Figure 15).

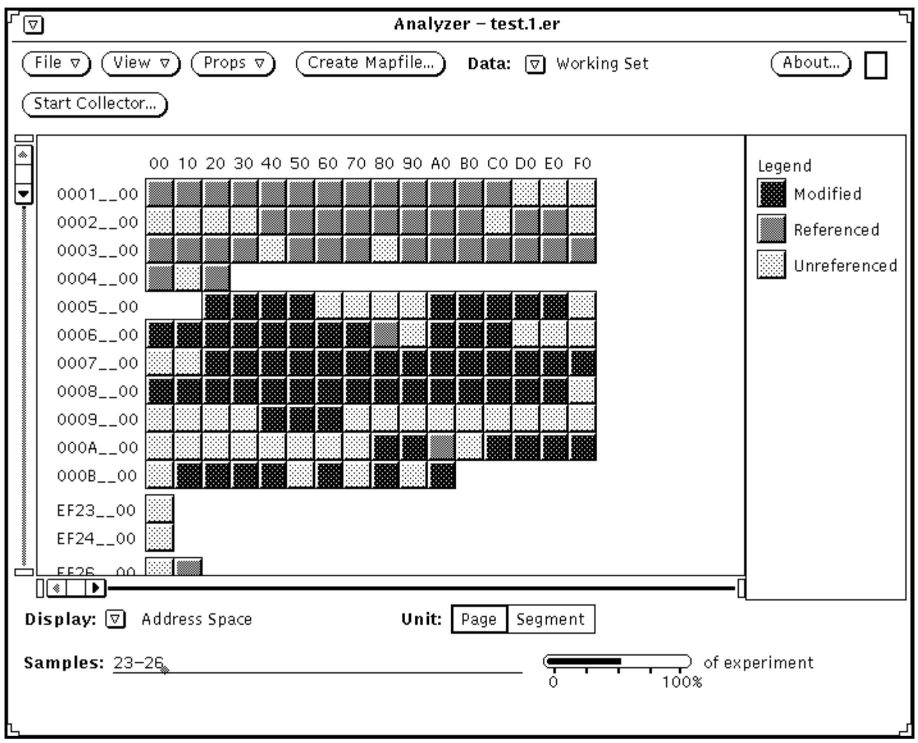


Figure 15 Pages Display

SPARC systems use pages that occupy either 4 or 8 Mbytes of memory. The Analyzer categorizes these pages into four classes:

- **Modified** — a *modified* page is written to during program execution.
- **Referenced** — a *referenced* page is read from during execution, but not written to.
- **Unreferenced** — an *unreferenced* page has not been has not been modified and has not been referenced.

The Pages display helps you identify memory that is most valuable to the application (modified and referenced pages) as well as memory that is unused because the experiment did not exercise all the program's functionality or because the program has dead code or memory allocation problems. The Pages

display offers two viewing modes: pages and segments. The Pages mode shows you individual pages; the Segment mode shows you individual segments.

Statistics

The Statistics display provides statistical information about an application's performance that is not obvious or visible in any of the other Analyzer displays.

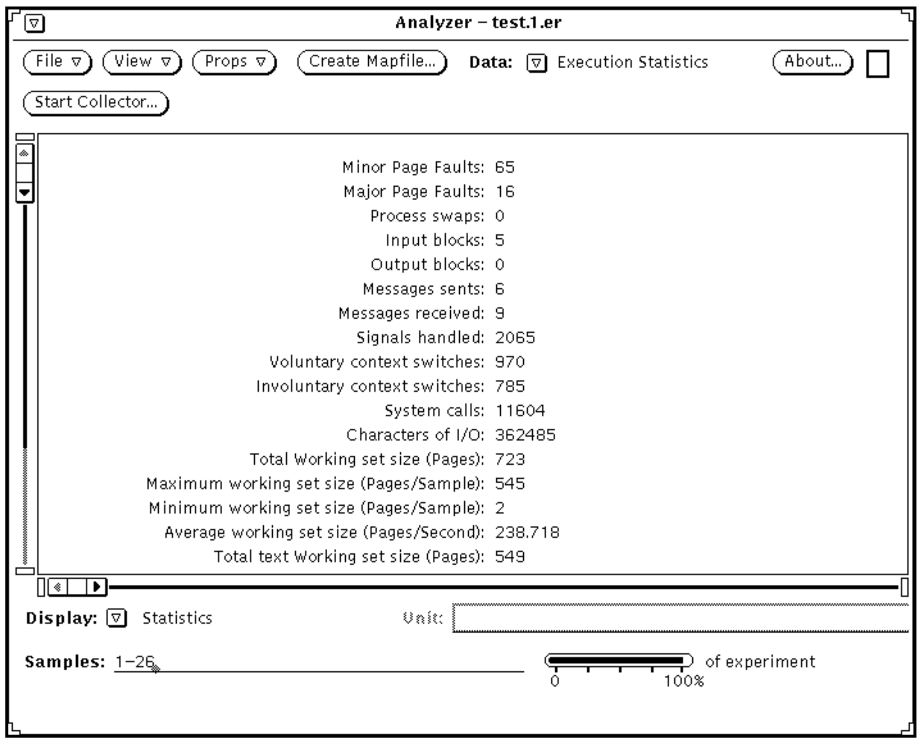


Figure 16 Statistics Display

The Statistics display often yields surprising information about system behavior that affects application performance. For example, the display shows the number of times a process is swapped out of main memory. A high number of process swaps may indicate that a large number of other processes were running at the time you collected experiment data or that the workstation was provided with too little memory. Although you cannot change these conditions directly by altering your application, they may affect the minimum system requirements you recommend for the application.

The following data is shown in the statistics display:

- **Minor page faults** — the number of page faults serviced that did not require any physical input/output (I/O) activity.

- **Major page faults** — the number of page faults serviced that did require physical I/O activity. Pages serviced include kernel-initiated page ahead operations. If this number is nonzero, then the overview display shows text or data fault wait time.
- **Process swaps** — the number of times a process was swapped out of main memory.
- **Input blocks** — the number of times a read system call was performed on a noncharacter or special file.
- **Output blocks** — the number of times a write system call was performed on a noncharacter or special file.
- **Messages sent** — the number of messages that were sent through sockets.
- **Messages received** — the number of messages that were received through sockets.
- **Signals handled** — the number of signals handled. Although signals are not visible in the other displays, the routines that handle signals are visible. Therefore, large signal handling activity in the statistics display alerts you to examine functions that handle signals and are shown to use large amounts of time in the histogram display.
- **Voluntary context switches** — the number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed in order to wait for availability of a resource.
- **Involuntary context switches** — the number of times a context switch resulted because the process was preempted by a higher priority process or because the process exceeded its time slice.
- **System calls** — the total number of system calls produced by the process.
- **Characters of I/O** — the number of characters transferred in or out of a process by read and write calls to a character device or file.
- **Working set size** — total number of memory pages allocated to the working set.
- **Maximum working set size** — the maximum number of memory pages used by the working set in any one sample.
- **Minimum working set size** — the minimum number of memory pages used by the working set in any one sample.
- **Average working set size** — the number of memory pages used by the working set while the program was running, averaged over the total number of samples.

Reordering Program Text

You can reorder an application to reduce the text working set size. The Analyzer lets you create a *mapfile*, which specifies a function ordering that reduces the size of the text working set. Once you have created the mapfile, you can use it to rebuild the program. The default reordering strategy is based on profiling data and function sizes sorted in descending order. The strategy of ordering functions in descending order is to reduce the text working set size.

Reduction occurs because infrequently used functions are grouped on a page and are rarely part of the text working set. All functions not listed in the mapfile are placed after the listed functions.

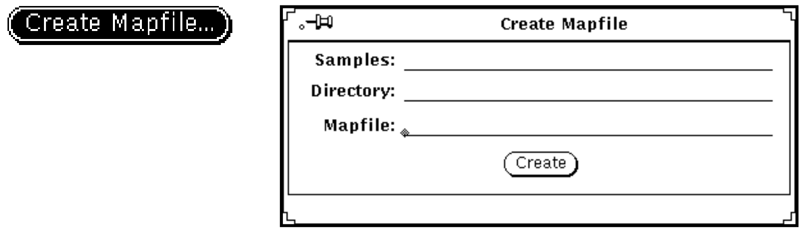
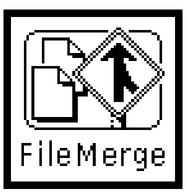


Figure 17 Reordering Program Text

Exporting Experiments

You can export the data collected by the Debugger into files for use by other programs such as spreadsheets or custom-written applications. The format of the export data file is documented in SPARCworks technical documentation.

FileMerge



SPARCworks FileMerge provides a convenient way to merge two text files or directories of files. Most FileMerge functionality is duplicated in the Solaris `diff`(1) utility, but the FileMerge window interface helps merge files more easily and quickly than any command-line based tool.

In addition to loading two files to be merged, a user can specify a third file, called the *ancestor* of the two files. The two files to be merged are called *descendants* of the ancestor. When an ancestor file has been specified, FileMerge marks lines in the descendants that are different from the ancestor and produces a merged file based on all three files.

FileMerge Window

The graphical interface for FileMerge consists of one main window, in which users do most of their work, and two pop-up windows for handling files and settings properties.

The FileMerge window is shown in Figure 18. The left and right text panes at the top show input files to be merged in read-only form; the text pane at the bottom is the output file — an editable, merged version of the two input files.

Users with color monitors can select colors from a palette to highlight the various line differences. To bring up the color palette, select the Colors category from the FileMerge Properties window.

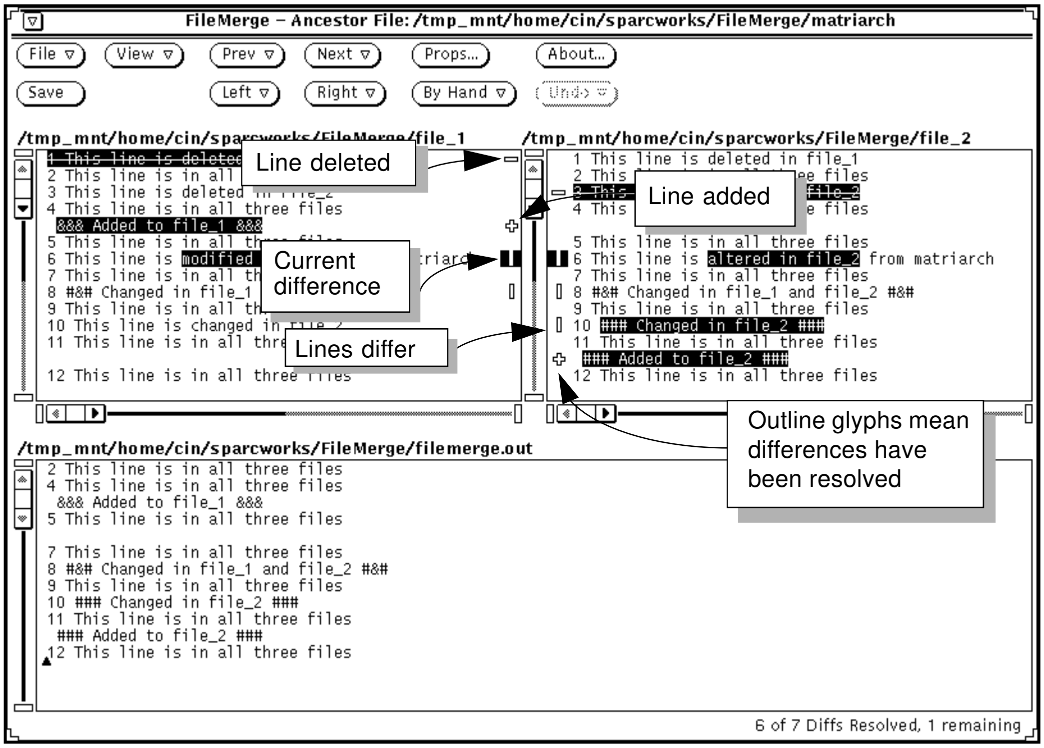


Figure 18 FileMerge Window with Loaded Files and Common Ancestor

Three Input Files

When an ancestor file has been specified for the two files to be merged (as shown in Figure 18), lines in each descendant are marked according to their relationship to the corresponding lines in the common ancestor:

- If a line is identical in all three files, then no glyph is displayed.
- If a line is not in the ancestor but was added to one or both of the descendants, then a plus sign glyph (+) is displayed next to the line in the file where the line was added and the different characters are highlighted.
- If a line is present in the ancestor but was removed from one or both of the descendants, then a minus sign (-) is displayed as a placeholder in the file from which the line was removed and the different characters are highlighted and in strikethrough.
- If a line is in the ancestor but has been changed in one or both of the descendants, then a vertical bar glyph (|) is displayed next to the line in the file where the line was changed, and the different characters are highlighted.

Resolved differences are marked by glyphs in outline font.

Two Input Files

When two files have been loaded without an ancestor file, lines in each file are marked by glyphs to indicate when they differ from corresponding lines in the other file:

- If two lines are identical, no glyph is displayed.
- If two lines are different, a vertical bar glyph (|) is displayed next to the line in each input text pane, and the different characters are highlighted.
- If a line appears in one file but not in the other, a plus sign glyph (+) is displayed next to the line in the file where it appears, and the entire line is highlighted.

Merging Files

By default, FileMerge constructs the merged output file by placing in it all lines that are common between the two input files. When a line is different between the two files, the user accepts a line from one of the files and places it in the output file. The user indicates a preference for one file or the other by clicking on control-panel buttons. As each difference is resolved, FileMerge advances automatically to the next difference.

If neither input file contains a suitable line to use in the output file, the user can edit the output file directly. In some cases, the differences from one input file will always be preferred over the other. In these cases, the difference lines from one or the other input file can be placed in the output file with a single control-panel button click.

Viewing Differences Read-Only

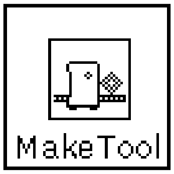
In some cases you may want to display source files in read-only mode. In these cases, FileMerge does not display a merged version of the files but only the loaded source files in the left and right panes. The second row of control buttons, which ordinarily govern how differences are merged into an output file, are also hidden in read-only mode.

Loading Lists of Files

The command line interface to FileMerge lets users specify lists of files to load sequentially. This capability is very useful when two entire directories must be merged. A list of ancestor files can also be specified.

To load files from a list, the files to be merged must be pairs of files with identical local names. The files themselves must be stored in separate directories. You then create a list of file names in a *listfile* and start FileMerge from the command line, specifying the name of the listfile, the two directories to be merged, an output directory, and optionally a third directory of ancestor files. You can then automatically load files successively as you finish resolving differences in each file pair.

MakeTool

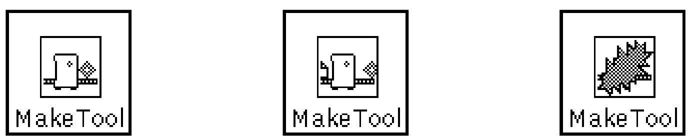


MakeTool provides a graphical user interface to `make`(1), the utility that oversees program compilation and ensures that programs are compiled from the newest sources. In addition, MakeTool contains a browser that helps you interpret makefiles by expanding macros and rules.

MakeTool Icons

MakeTool in its closed form is represented by one of three icons, as shown in Figure 19. Each icon represents a different condition of a `make` build process. A glance at the icon tells you the status of a long `make` task.

- The first icon indicates that a `make` has successfully completed. This icon is also displayed before the first `make` of a session.
- The second icon shows that `make` has been started but is not yet completed. This icon is animated: It shows source files being fed into a "`make` machine" and rolling out on a conveyor belt as compiled objects.
- The third icon indicates that `make` has failed due to an error.



Successful Make **Make In Progress** **Failed Make**

Figure 19 MakeTool Icons

MakeTool Window

The MakeTool base window (Figure 20) opens when MakeTool starts. MakeTool loads a makefile automatically using the same search algorithm as `make`. If MakeTool finds a makefile, it is loaded and displayed in the main MakeTool window. The fully qualified name of the makefile is displayed on the Makefile text field.

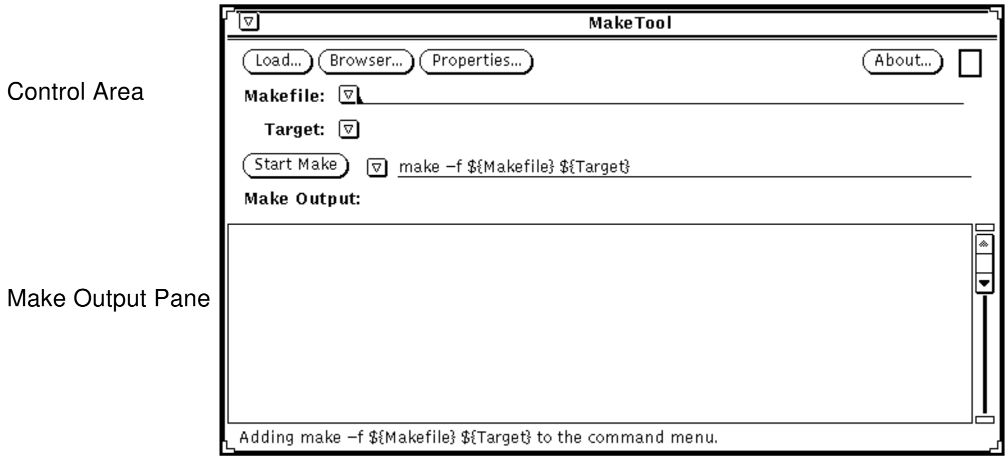


Figure 20 MakeTool Base Window

To build a `make` target, the user clicks on the Start Make button. MakeTool responds by starting a subshell, changing directories to the makefile's working directory (if necessary), and issuing a `make` command.

You can issue commands from the Start Make text field. MakeTool maintains a history of commands. All commands MakeTool issues are echoed in the Make Output pane in the MakeTool window. The Make Output pane is a read-only window.

Building the Make Menu

The abbreviated button next to the Target button reveals a menu that is filled with arguments. As MakeTool loads a makefile, it examines the makefile to find high-level `make` targets and places them in the menu. Subsequently, the user can initiate a build of one of these targets simply by selecting from the menu. Because the targets are displayed in a menu, users do not need to list the text of a makefile in order to identify the targets they want to build, as is often the case when they use `make` directly from the command line.

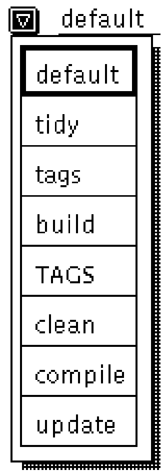


Figure 21 Target Abbreviated Menu

Targets that are not in the Target menu can be built by typing them in place of `${Target}` the default `make` command on the Start Make text entry field (see Figure 20).

Makefile Browser

As developers have learned to exploit the power of `make` and apply it to larger and larger projects, makefiles have grown more complex and difficult to maintain. Macros in particular can be difficult to understand because they are often nested, appended to, and defined conditionally (so that they take on different values depending on the target being compiled). MakeTool contains a browser to help developers interpret makefiles by expanding rules and macros. Note that the makefile browser shows rules and macros but does *not* provide the means to edit them.

Browser Window

Open the Browser window by clicking on the Browser button in the MakeTool base window. The window is shown in Figure 22.

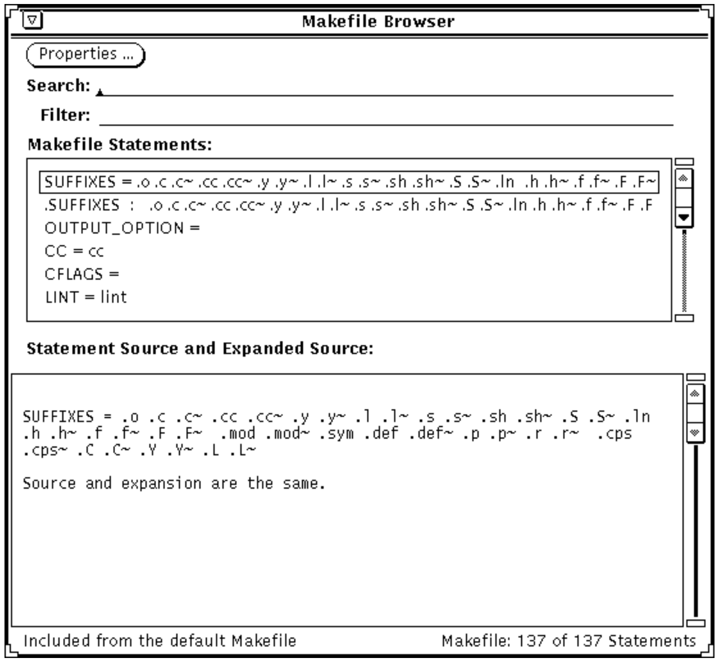


Figure 22 Makefile Browser

Rules and Macros

The Makefile Browser displays all the rules and macros in the makefile in a single list. Rules and macros are defined as follows:

- **Rule** — a makefile *rule* is defined as a target, its dependencies, and the method used to build the target from the dependencies. Typically, a rule is a command line in a makefile that invokes a compiler or linker.
- **Macro —** a makefile *macro* is a variable that is used in dependency lists and rules. Macros are used in makefiles to increase their flexibility and maintainability.

  In makefiles, any line of the form `STRING1 = string2` is a macro definition. `STRING1` is the name of the macro, and `string2` is the definition of the macro. Following such a definition, whenever `make` encounters the expression `$(STRING1)` in the makefile, it expands the string to the value `string2`.

Rules can be sorted into three categories for display purposes: special rules, implicit rules, and ordinary rules defined within the current makefile. Similarly, macros can be sorted into two categories: conditional macros and all others.

You can narrow the range of statements displayed in the Browser by setting filters in a Properties window (Figure 22). The filters allow only certain statements to be shown — only rules used to make the current target, for example — so that confusing, extraneous statements are hidden from view.

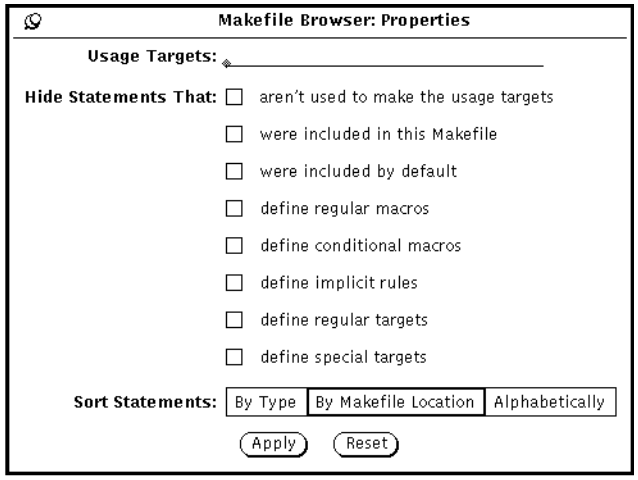


Figure 23 Makefile Properties Window

Within each category of rule or macro, the Browser sorts statements into the following types:

- Special rules first, then regular rules, then implicit rules
- All macros that are *not* conditional macros first, followed by conditional macros

Viewing Statement Sources and Expansions

As shown in Figure 22, each statement is displayed on a single line in the Makefile Statements pane. Below the Makefile Statements pane, the complete source of a selected statement (its literal representation in the makefile) and its expanded version are shown in a separate pane.

Statement Sources

Seeing the source of a statement is useful when the entire rule or macro is too long to be shown on a single line; the display of source wraps the line so that all of it is visible.

Statement Expansions

To expand a statement, MakeTool replaces all macro expressions of the form `$(`*NAME*`)` with their values. Because many macros are made up of other macros, determining their expanded values from source statements can be difficult for a user without the aid of MakeTool Browser.

Searching for Text

The Browser window provides a text entry field for entering search strings. In Figure 24, a user has typed the word “Sun” into the search text entry field. The first line that contains the word is automatically selected in the Makefile Statements pane. To select other lines that contain “Sun,” the user would press the down-arrow key. The search *wraps* through the makefile, so repeated pressing of the down-arrow key would eventually bring the user back to the first line that contains "Sun."

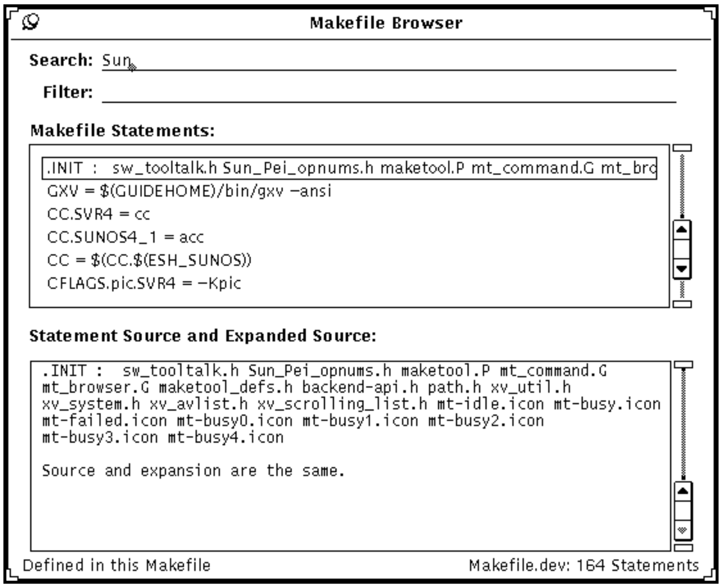


Figure 24 Browser Window with "Sun" Search String

Filtering Statements

The Browser window also provides a text entry field for entering filter characters. When text is entered in the Filter field, only statements that contain the text are displayed.

Figure 25 shows a Browser window in which the user has set a filter for the string "Pei" and combined it with a search for the word "Sun." The Makefile Statements pane shows only statements that contain the string "Pei." From those statements, the Browser searches for statements that contain the word "Sun."

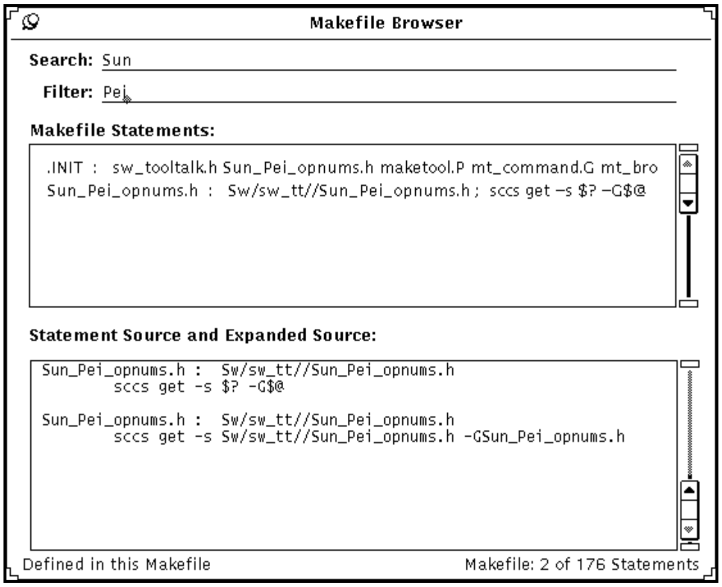


Figure 25 Browser Window with "Sun" Search String and "Pei" Filter

Conclusion

SPARCworks tools, with their high performance and full feature set, deliver a rich software development environment. Together, SPARCworks tools provide the following advantages:

- The consistent look and feel provided by the standard graphical user interface shortens learning time for new users.
- Visual and functional compatibility with DeskSet™ tools such as File Manager and Text Edit give SPARCworks tools added value.
- SPARCworks Manager provides a uniform way to integrate new tools into the SPARCworks toolset and coordinate them with the standard SPARCworks tools.
- For simultaneous static and dynamic program analysis, SourceBrowser and Debugger form a powerful combination.
- The concise graphical presentation of data provided by SourceBrowser and Analyzer quickly clarifies program structure and uncovers performance bottlenecks.
- The process of building programs and maintaining makefiles is assisted by MakeTool, a graphical interface for the popular `make` utility.
- Differences between source files can be found easily and a merged version of two source files can be created quickly with FileMerge.

Further Reading

The following SunPro white papers provide more information about the Sun software development environment.

Solaris OpenWindows V3 Collection, PN 91021-0 1.5K

OpenWindows Developer's Guide: A User Interface Builder

TeamWare Solutions Guide, PN TWSG-0194/1K

Creating Multithreaded/Multiprocessing Applications

Integrating Applications with the SPARCworks 3.0 Toolset PN 801-7208-01

SunPro Product Guide PG 0194

SunPro
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
MS UMPK03-205
Mountain View, CA 94043-1100 U.S.A.

Information subject to change without notice. Revision 2.0 1/94