Introduction to Sun WorkShop

Please
Recycle

Adobe PostScript™

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

■ Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.

■ Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Figures

Tables

Code Examples

Preface

Introduction to Sun WorkShop acquaints you with the basic program development features of the Sun WorkShop™ integrated programming environment. This book is intended for application developers who have a working knowledge of Fortran, C, or C++ and Solaris/UNIX® and who want to understand the main development features of Sun WorkShop.

To get updated information about Sun WorkShop through the Web, use the Web Updates Dialog Box. To open the Web Updates dialog box, choose Help ➤ Web Updates in any Sun WorkShop window.

Multiplatform Release

This Sun WorkShop release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* and Solaris *Intel Platform Edition* Operating Environments.

Note – The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture, whereas "*Intel Platform Edition*" appears in the product name.

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing** `/opt/SUNWspro/bin/`.

If you find the paths, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your `PATH` environment variable by following the instructions in this section.

To determine if you need to set your `MANPATH` environment variable:

1. **Request the `workshop` man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

If the `workshop`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your `MANPATH` environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the `/opt` directory. Contact your system administrator if your Sun WorkShop software is not installed in `/opt`.

The `PATH` and `MANPATH` variables should be set in your home `.cshrc` file if you are using the C shell or in your home `.profile` file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your `PATH` variable:

  `/opt/SUNWspro/bin`

■ To access Sun WorkShop man pages with the `man` command, add the following to your `MANPATH` variable:

```
/opt/SUNWspro/man
```

For more information about the `PATH` variable, see the `csh(1)`, `sh(1)`, and `ksh(1)` man pages. For more information about the `MANPATH` variable, see the `man(1)` man page. For more information about setting your `PATH` and `MANPATH` variables to access this release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

How This Book Is Organized

Introduction to Sun WorkShop contains the following chapters and appendixes:

Chapter 1, "About Sun WorkShop," is an overview of the Sun WorkShop programming environment.

Chapter 2, "Getting Started," describes how to begin working in the Sun WorkShop integrated programming environment.

Chapter 3, "Building Programs," describes building an application with Sun WorkShop and how to identify build errors.

Chapter 4, "Debugging a Program," highlights the debugging service offered in Sun WorkShop.

Chapter 5, "Browsing Source Code," describes source code browsing, call graphing, class graphing, and class browsing.

Chapter 6, "Analyzing Program Performance," gives an overview of the Sampling Collector and Sampling Analyzer.

Chapter 7, "Merging Source Files," describes how to compare different versions of a source file and merge the changes.

Appendix A, "Sun WorkShop and Text Editor Resources," describes some of the resource settings in Sun WorkShop that you can edit.

Appendix B, "The `make` Utility and Makefiles," describes options that you can set for the `make` utility.

Appendix C, "The `dmake` Utility," describes how the distributed make (`dmake`) utility distributes builds over several hosts to build programs concurrently over a number of workstations or multiple CPUs.

Appendix D, "Source Browsing With `sbquery`, `sb_init`, and `sbtags`," describes the use of the command line to complete browsing tasks that are also available within the Sun WorkShop integrated programming environment.

The Glossary defines the terms used in this book.

Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1   Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% `**`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the** `docs.sun.com`[sm] **Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at `http://docs.sun.com`.

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:

  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.

  - In your Netscape™ Communicator 4.0 or compatible Netscape version browser, open the following file:

    `/opt/SUNWspro/docs/index.html`

  (If your Sun WorkShop software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.
Forte Developer 6 / Sun WorkShop 6	*Analyzing Program Performance With Sun WorkShop 6*	Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool `er_print`, the LoopTool and LoopReport utilities, and UNIX profiling tools `prof`, `gprof`, and `tcov`.
	Debugging a Program With dbx	Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.
Forte™ C 6 / Sun WorkShop™ 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte™ C++ 6 / Sun WorkShop™ 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop™ 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.

TABLE P-3   Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte™ TeamWare 6 / Sun WorkShop TeamWare 6	*Sun WorkShop TeamWare User's Guide*	Describes how to use the Sun WorkShop TeamWare code management tools.
Forte Developer 6/ Sun WorkShop Visual 6	*Sun WorkShop Visual User's Guide*	Describes how to use Visual to create C++ and Java™ graphical user interfaces.
Forte™ / Sun Performance Library™ 6	*Sun Performance Library Reference*	Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.
	Sun Performance Library User's Guide	Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.
Numerical Computation Guide	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ Class Library Reference*	Provides details on the `Tools.h++` class library.
	Tools.h++ User's Guide	Discusses how to use the C++ classes to enhance the efficiency of your programs.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	Document Title	Description
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

About Sun WorkShop

The Sun WorkShop programming environment simplifies complex development tasks by providing an integrated development environment for building, editing, debugging, source browsing, and tuning your C++, C, and Fortran 77/95 software development projects.

The Sun WorkShop integrated programming environment includes:

- Integrated text editors
- Compilers
- An integrated debugging service
- A source code browser
- Performance, source code management, and GUI-building tools

Integrated Text Editors

Text editors are the center of the Sun WorkShop integrated development tool set that includes building, debugging, and browsing. The Sun WorkShop programming environment makes it possible to evaluate expressions, set breakpoints, and step through functions from your text editor.

This release of Sun WorkShop provides the following integrated editors:

- **NEdit, version 5.0.2.** A graphical user interface-style plain-text editor for X/Motif systems. NEdit is the editor that Sun WorkShop uses by default (to change your text editor, see "Choosing a Text Editor and Text Editor Options" on page 20). For more information about NEdit, see "NEdit Editor Window" in the Text Editing section of the online help and the NEdit web page at `http://www.nedit.org`.

- **XEmacs, version 20.4 or compatible versions.** A customizable text editor and application development system. For more information, see "XEmacs Editor Window"in the Text Editing section of the online help and the XEmacs web page at `http://www.xemacs.org`.

- **GNU Emacs, version 19.28 or compatible versions.** An extensible, customizable, self-documenting real-time display editor. For more information, see "GNU Emacs Editor Window" in the Text Editing section of the online help and the GNU web page at `http://www.gnu.org`.

- **Vi.** A screen-based editor on UNIX systems. For more information, see "Vi Editor Window" in the Text Editing section of the online help.

- **Vim, version 5.3 with graphical user interface option.** An improved version of the vi standard text editor on UNIX systems. For more information, see "Vim Editor Window"in the Text Editor section of the online help or the Vim web page at `http://www.vim.org`.

Note – Not all text editors are available in all locales.

For more information about the Sun WorkShop editors, see:
- "Choosing a Text Editor and Text Editor Options" on page 20
- The Text Editing section in the online help

Compilers

This release of Sun WorkShop supports the following compilers:

- Sun WorkShop™ Compilers C++ (Available with Sun Visual WorkShop™ C++)

  The C++ compiler (`CC`) supports the ISO standard for C++, ISO IS 14882:1998, Programming Language C++. The following requirements in the standard are not supported in this release: Templates as template parameters and universal character names. For more information about the C++ compiler, see the *C++ User's Guide.*

- Sun WorkShop™ Compilers C

  This compiler is fully compliant with the ANSI C language and environment standard, and it also supports traditional K&R C. The C optimizer provides significant performance increases over nonoptimized code. The code optimizer removes redundancies, efficiently allocates registers, and schedules instructions. Also featured is an incremental linker to reduce link time during the debugging phase. For more information about the C compiler, see the *C User's Guide.*

- Sun WorkShop™ Compilers Fortran (Available with Sun Performance WorkShop™ Fortran)
  - Fortran 95

    This release is a complete implementation of the Fortran 95 ISO/IEC 1539:1997 standard. This standard has added many features, such as an improved ability to express mathematical formulas more directly in the programming language. In addition, the Fortran 95 compiler works with the rest of Sun WorkShop to automatically parallelize your code. For more information about the Fortran 95 compiler, see the *Fortran User's Guide* or *Fortran Programming Guide.*
  - Fortran 77

    This compiler is a complete implementation of the Fortran 77 ANSI X3.9-1978, ISO 1539-1980 standards. It has an improved ability to express mathematical formulas more directly in the programming language, as well as extensions that provide compatibility with VAX VMS Fortran and Cray Fortran. For more information about the Fortran 77 compiler, see the *Fortran User's Guide* or *Fortran Programming Guide.*

Integrated Debugging Service

Sun WorkShop uses a window-based source code debugging service that provides the ability to run a program in a controlled fashion and to inspect the state of a stopped program. You can perform most debugging operations from the Debugging window and the windows accessed from it. You can also perform basic debugging operations from a text editor window containing the source code, which opens automatically when you load a program for debugging. You have complete control of the dynamic execution of a program, including the collection of performance data. A line-oriented, source-level debugger called dbx is also included with Sun WorkShop.

For more information, see:

- Chapter 4
- The Using the Debugging Window section of the online help
- The Using dbx Commands section of the online help

Source Code Browser

You can browse source code written in C, C++, and Fortran 77/95 by issuing a query in the Browsing window in either pattern search mode or source browsing mode. Pattern search mode allows you to search your source code for any text string, including text embedded within comments. Source browsing mode allows you to find all occurrences of any program-defined symbol in your code by searching in a database that is generated when Sun WorkShop compiles your source files with a source browsing option. (One of the selections you can make when you are creating or editing a project is for Sun WorkShop to generate the database when it compiles your code.) You then view the occurrences or matches to your query with their surrounding source code in the Browsing window match pane.

You can also graph the function and subroutine relationships in your program, and if your source code is written in C++, you can browse and graph the classes defined in your program.

For more information, see:
■ Chapter 5
■ The Browsing Source Code section of the online help

Performance, Source Code Management, and GUI-Building Tools

By default, the Sun WorkShop main window provides access through the Tools menu to the Sampling Analyzer, which helps you analyze your program performance, and Merging, which is part of Sun WorkShop TeamWare source code management tools. If you have Sun Performance WorkShop Fortran, the Tools menu also provides access to the multithreaded tool LoopTool. If you have Sun Visual WorkShop C++, you also have access to Visual, the GUI-building tool.

Multithreaded Development Tools

Sun WorkShop includes tools for developing multithreaded applications. Sun WorkShop Debugging supports dynamic analysis and control of multithreaded programs. LockLint analyzes source code for potential synchronization errors, such

as deadlock and data race conditions. LoopTool displays a graph of loop runtimes and shows which loops were parallelized. Together they provide support for multithreaded program development.

For more information, see:

- Chapter 6
- *Analyzing Program Performance With Sun WorkShop 6*
- The Analyzing Program Performance and Analyzing the Loops in Your Program sections of the online help
- "Multithreaded Program Debugging" in the Using the Debugging Window section of the online help

Sun WorkShop TeamWare

Sun WorkShop TeamWare source code management tools allow you to manage source code files through a set of GUIs or from the command line. Sun WorkShop TeamWare allows a team to work in parallel at different sites to coordinate, integrate, and build a product. The services include:

- **Configuring.** This manages and integrates source code configurations and releases.
- **Versioning.** This creates and tracks file version histories.
- **Freezepointing.** This baselines a software configuration or release for later retrieval.
- **Building**. This executes build jobs on multiple Solaris hosts.
- **Merging.** This merges source files and coordinates source changes.

For more information:

- See Chapter 7 in this book.
- See the *Sun WorkShop TeamWare User's Guide.*
- Choose Help from the TeamWare Configuring window menu bar.

Sun WorkShop Visual

(Available with Sun Visual WorkShop C++)

Visual helps you design graphical user interfaces (GUIs), generate portable object-oriented code, and develop Motif, Java, or Microsoft Foundation Class GUIs. Visual automatically generates the code when the design is complete.

For more information, see the *Sun WorkShop Visual User's Guide.*

Getting Started

Once Sun WorkShop is installed and added to your command path (see "Access to Sun WorkShop Development Tools" on page 2), you can start Sun WorkShop by typing the following at a command line:

```
% workshop&
```

For more information about the workshop command, see the workshop(1) man page.

This chapter describes how to begin working in the Sun WorkShop programming environment and contains basic information about:

■ Working with projects
■ Using the WorkShop main window

For step-by-step instructions and more information, see the Sun WorkShop online help (you can access the online help through the Help menu in any Sun WorkShop window).

Working With Projects

This release of Sun WorkShop uses projects to keep track of the files, programs, and targets associated with your development projects and to build your programs without your needing to write a makefile. A project is a list of files and compiler, debugger, and build-related options used to build an executable, a static library/ archive, a shared library, a Fortran application, a complex application, or a user makefile application.

When you start Sun WorkShop, the Welcome to Sun WorkShop dialog box opens and gives you immediate access to Sun WorkShop projects and the project wizard. Click on the "projects" link in the description pane to access information about projects in the online help. You can also click Help in the Welcome to Sun WorkShop dialog box for more information about that dialog box.

Through the Welcome to Sun WorkShop dialog box or the commands available from the Project menu in the WorkShop main window, you can:

- Create a new project or build a simple program using the project wizard and your own makefile or a makefile Sun WorkShop creates for you (see "Creating a New Project" in the Working With Projects section of the online help)
- Change existing project settings, including how you want your project compiled and whether you want source browsing information generated (see "Editing a Project" and "Edit Current Project Window" in the Working With Projects section of the online help)

If you have Sun WorkShop worksets, you can automatically convert your worksets to Sun WorkShop 6 projects when you load them (for step-by-step instructions, see "Converting a Workset to a Project" in the Working With Projects section of the online help).

You can also choose to use the Sun WorkShop 6 programming environment without loading a project. Picklists keep track of the files, programs, directories, and targets associated with your development projects (see "WorkShop Targets" in the Building Programs section of the online help for more information). You can access each file, build target, and debug executable from the menus in the WorkShop main window.

Creating a Project

Through the Welcome to Sun WorkShop dialog box or through the Project menu in the WorkShop main window, you can ask the project wizard to help you create a project.

From the Welcome to Sun WorkShop dialog box, you can:

Create a New Project	The Create a New Project wizard guides you through creating a new project from existing source files. You choose the type of project, and the wizard prompts you for information to create that type of new project.
Build a Simple Program	The Create New Project wizard helps you build a single executable from a set of source files. There are limited compilation options and only standard libraries to which to link.

Create an Empty Project	The Create Empty Project dialog box opens, and you request that a new project be created that has no existing source files. A text editor window opens so you can begin creating your program.

The Create New Project wizard prompts you to define your project settings. Then Sun WorkShop creates the type of project you defined (a file with a `.prd` file extension) with the source files you requested.

You can share that project file information with multiple members of your development team by creating a project that has an absolute/full path to the project file and a relative (file/base name only) project directory. For example, your team has a workspace for your project, and you create a project named `ws.prd` in the top level directory of the workspace:

```
Project file name: /home/workspaces/ws/ws.prd
Project directory: .
```

A team member can have a copy of the workspace in `/export/myws` and by opening the project file `/export/myws/ws.prd`, the project applies to that team member's local files and to no other team member's files.

Building Project Targets

Once you have created your project, you can build project targets by doing one of the following:

- Click Build in the WorkShop main window tool bar.
- Choose Build ➤ Build Project from the WorkShop main window menu bar.
- Use the `makeprd` command at the command line (for example, to build your project from the command line as part of a script or cron file).

After you select one of these methods, Sun WorkShop:

1. Creates a makefile from the project settings you defined in the Create New Project wizard or the Edit Current Project window.

2. Opens the Building window.

3. Starts the `make` utility.

4. Shows the results of the build in the Building window.

When you click on a build error hypertext link in the Building window, your text editor opens with the build error highlighted so you can examine and fix it.

For more information, see:

- Chapter 3

- The Building Programs section of the online help
- The `makeprd`(1) man page

Editing a Project

You can edit your project through the Edit Current Project window. To open the Edit Current Project window, choose Project ➤ Edit Project in the WorkShop main window. For more information, see "Editing a Project" and "Edit Current Project Window" in the Working With Projects section of the online help.

Using the WorkShop Main Window

The WorkShop main window helps you access the tools you need to create, develop, debug, and fine tune your applications and lets you choose your text editor and set different types of options.

FIGURE 2-1 displays the WorkShop main window.

FIGURE 2-1   WorkShop Main Window

For more information about the WorkShop main window, see "WorkShop Main Window" in the online help. (To open the online help, choose Help ➤ Contents in the WorkShop main window.) To set colors, fonts and other types of resources used in WorkShop windows, see Appendix A.

Choosing a Text Editor and Text Editor Options

This release of Sun WorkShop provides the following integrated editors:

- NEdit, version 5.0.2 (the Sun WorkShop default editor)
- XEmacs, version 20.4 or compatible versions
- GNU Emacs, version 19.28 or compatible versions

- Vim, version 5.3 with graphical user interface option
- Vi

To change your default editor and set text editor options, choose Options ➤ Text Editor Options in the WorkShop main window. The Text Editor Options dialog box opens. The options displayed in the dialog box change depending upon the editor you choose in the Editor to Use pull-down menu. The editor you choose will remain your default editor until you select another editor in the Text Editor Options dialog box.

For more information about each editor's options, see:

- The online documentation available from the Help menu in the editor's menu bar
- "Text Editor Options Dialog Box" in the Text Editing section of the WorkShop online help

You can set colors, fonts and other types of resources used in WorkShop integrated text editors. For information on changing the default resources of Sun WorkShop in the Common Desktop Environment (CDE) and non-CDE environments, see Appendix A.

Setting Startup and Project Options

Sun WorkShop offers you startup and project options through the Options menu in the WorkShop main window.

Startup Options

By default, Sun WorkShop at startup:

- Remembers the size and position of its windows from your previous Sun WorkShop session and redisplays them
- Shows the Sun WorkShop splash screen
- Shows the Welcome to Sun WorkShop dialog box

To change these startup options, choose Options ➤ Startup Options from the WorkShop main window menu bar to open the Startup Options dialog box. For more information about these options, see "Startup Options Dialog Box" in the WorkShop Main Window section of the online help.

Project Options

By default, Sun WorkShop treats projects in the following way:

- At startup, Sun WorkShop opens the last project you had open and populates your menu picklists with the items contained in that project.
- When you exit Sun WorkShop or open another project, Sun WorkShop prompts you to save or discard project changes instead of saving automatically.
- When you exit Sun WorkShop or open another project, Sun WorkShop automatically saves your menu picklist entries on all your menus.
- Sun WorkShop sets the maximum number of menu picklist entries at 20.
- Sun WorkShop uses the directory from which it was started as the default directory for its tools.

To change these project options, choose Options ➤ Project Options from the WorkShop main window menu bar to open the Project Options dialog box. For more information about these options, see "Project Options Dialog Box" in the WorkShop Main Window section of the online help.

Accessing Sun WorkShop Components

The WorkShop main window helps you access the tools you need to create, develop, debug, and fine tune your applications. The following chapters introduce you to the components of Sun WorkShop and how Sun WorkShop can help you:

- Build your development projects
- Debug your programs
- Browse your code
- Analyze your program's performance
- Merge your source files

Building Programs

Sun WorkShop projects allow you to customize your builds, build your programs without having to write a makefile, or build a program using your own makefile. You can also build an application without a current project, run one build job or several build jobs concurrently, and fix build errors using the Building window and the Sun WorkShop editor of your choice.

For step-by-step instructions and more information, see the Building Programs section of the online help (you can access the online help through the Help menu in any Sun WorkShop window).

Working With Targets

When building in Sun WorkShop, two types of targets are involved:

- Sun WorkShop targets
- User makefile targets

Sun WorkShop Target

A Sun WorkShop target is an object derived from the build directory, the build command, the makefile, and the make target:

- **Build directory.** The directory from which the build process is started and also the default directory for the makefile.
- **Build command.** The command that starts the `make` utility, which reads the makefile and builds the make targets.

- **Makefile.** A file that contains entries that describe how to bring a make target up to date with respect to those files on which it depends (called *dependencies*). Since each dependency is a make target, it may have dependencies of its own. Targets and file dependencies and subdependencies form a tree structure that make traces when deciding whether or not to rebuild a make target.

- **Make target.** An object that make knows how to build from the directions (rules) contained in a particular makefile. For example, a make target could be all or clean. Makefiles are generally designed so that the default target (the one you get when you do not specify a target) is the most commonly built target.

When a Sun WorkShop target is built, it is added to the picklist of Sun WorkShop targets in the Build menu and in the Build ➤ Edit Target command. When you begin a build, Sun WorkShop looks for the first target in the Sun WorkShop target list and builds it.

A project can contain multiple targets. For an executable, static library/archive, shared library, or Fortran application, your executable/library is one target, and a special Clean target is another (found in the Build menu picklist). The Clean target deletes all of your project's generated files (for example, the .o files), the source browsing database, the C++ templates database, the executable itself, and other build-related files.

For a complex project, you can have more targets, which are listed in the Build menu picklist. For example, your project can generate five libraries and an executable to link them together. Each library or executable is then a WorkShop target, and you can build each individual one by selecting it from the Build menu picklist.

User Makefile Target

A user makefile target is an object that make knows how to build from the directions (rules) contained in a particular makefile. Makefiles are generally designed so that the default target (the one you get when you do not specify a target) is the most commonly built target.

A makefile contains entries that describe how to bring a make target up to date with respect to those files on which it depends (called dependencies). Since each dependency is a make target, each dependency might have dependencies of its own. Targets and file dependencies and subdependencies form a tree structure that make traces when deciding whether or not to rebuild a make target.

For a user makefile project, each target listed in the Build menu picklist is a makefile or a makefile target to be built.

Using the Building Window

The Building window displays information on program compilation. You can open the window by choosing Build ➤ Show Building Window in the Sun WorkShop main window. FIGURE 3-1 shows the Building window.

From the Building window, you can:

- Start a build
- Stop a build in progress
- Edit build parameters
- Save the build output to another file
- View build errors
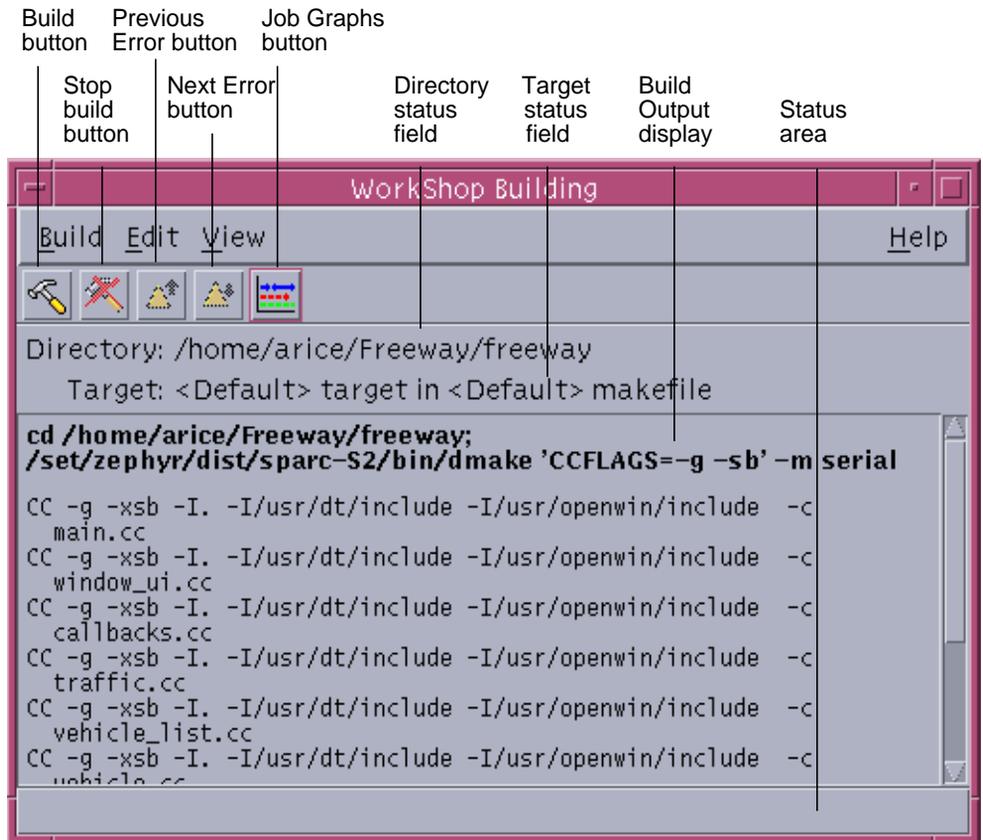
FIGURE 3-1 Building Window

Building a Program

You can build your entire project or only one of your project targets. After you ask Sun WorkShop to build your project targets, Sun WorkShop:

1. Creates a makefile from the project definitions you provided in the Create New Project wizard or the Edit Current Project Window.

2. Launches the `make` utility.

3. Opens the Building Window to show the results of the build.

For more information, see the following topics in the Building Programs section of the online help:

- "Building a Project"
- "WorkShop Targets"

You can also specify build parameters using the Define New Target and Edit Target dialog boxes. You use the Define New Target dialog box to specify a new WorkShop target and the Edit Target dialog box to modify an existing WorkShop target (the Define New Target and Edit Target dialog boxes are identical). FIGURE 3-2 shows the Define New Target dialog box.
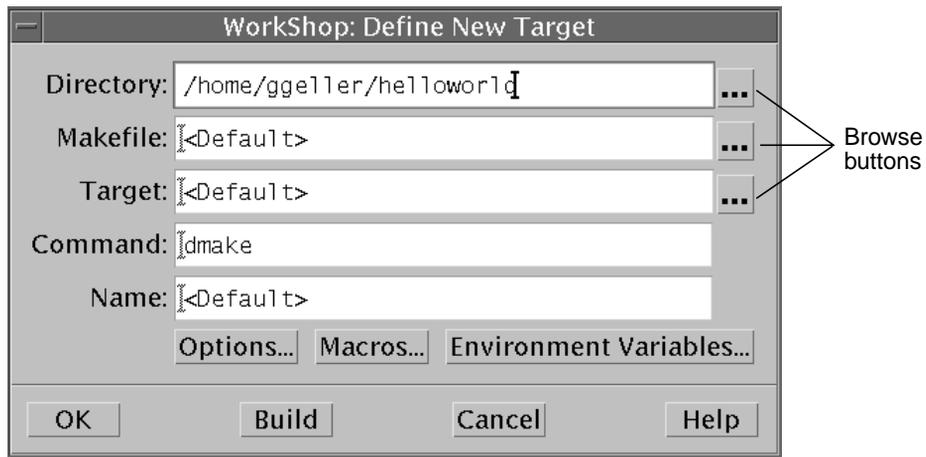
FIGURE 3-2 Define New Target Dialog Box

Building With Default Values

Sun WorkShop provides a default make target and a default make command (`dmake`), so you can begin a build without specifying a build command or a make target. You must still supply a makefile when you are building a user makefile project or when a project is not loaded (Sun WorkShop searches for a file named `makefile` or `Makefile` and allows `make` to figure out which one to use).

By using the project feature of Sun WorkShop, you can ask Sun WorkShop to create a makefile for you through the Create New Project wizard or the Edit Current Project window.

For more information, see "Building With Default Values" in the Building Programs section of the online help.

Specifying Your Own Build Values

If you have a makefile with a unique name, a certain make target, or a specific build command, you can define those build values in the Define New Target dialog box or Edit Target dialog box (this applies to a user makefile project or when a project is not loaded). For example, by specifying your own build command, you can filter out unnecessary warnings by passing `make` output through a filter. At a minimum, you must include a build directory. Sun WorkShop will then use the `make` command to find the makefile using `make`'s search order. See the `make`(1S) man page.

For more information, see "Specifying Your Own Build Values" in the Building Programs section of the online help.

Specifying Build Options

You can specify build options in the Build Options dialog box. To open the Build Options dialog box, click Options in the Edit Target dialog box. For information about the options available, click Help in the Build Options dialog box. When you are finished selecting the options you want, click OK in the Build Options dialog box. Then click Build in the Edit Target dialog box.

Before running a distributed build for the first time, you must create a `.dmakerc` runtime configuration file that specifies which machines are to participate as dmake build servers. The file contains groups (lists) of build servers and the number of jobs distributed to each build server. The `dmake` utility searches for this file on the dmake host to know where to distribute jobs. Generally, this file is in your home directory. If `dmake` does not find a runtime configuration file, it distributes two jobs to the local host. For information on setting up a runtime configuration file, see "The

.dmakerc File" in the Building Programs section of the online help and the `dmake`(1) man page. For more information about the `dmake` utility, see Appendix C and the `dmake`(1) man page.

To set up a machine to be used as a build server, you must create a configuration file called `/etc/opt/SPROdmake/dmake.conf` on the server's file system. Without this file, `dmake` refuses to distribute jobs to that machine. In the `dmake.conf` file, you specify the maximum number of jobs (from all users) that can run concurrently on that build server. See "The dmake.conf File" in the Building Programs section of the online help, Appendix C in this book, and the `dmake`(1) man page for more information.

Using Makefile Macros

You can specify makefile macros in the Make Macros dialog box (to open, click Macros in the Edit Target or Define New Target dialog box). Makefile macros let you refer conveniently to files or command options that appear in the description file. Through the Make Macros dialog box, you can add makefile macros to or delete them from the Persistent Build Macros list in your WorkShop target and then reassign values for makefile macros in the list. You can also add macros currently defined in the makefile to the list and override their values. For more information, click Help in the Make Macros dialog box, and see Appendix B for information about defining macros.

Using Environment Variables

You can specify environment variables for your build in the Environment Variables dialog box (to open, click Environment Variables in the Edit Target or Define New Target dialog box). Using the Environment Variables dialog box, you can add environment variables to or delete them from the Persistent Environment Variables list in your WorkShop target and reassign values for environment variables in the list. When you start the build, `setenv` commands for these environment variables are prepended to the build command. For more information, click Help in the Environment Variables dialog box.

Identifying Build Errors

When a build fails, the build errors display in the Build Output display pane of the Building window (see FIGURE 3-3). The location of the error is underlined and highlighted to denote a hypertext link to the location of the error in a source file.

Each error gives the name of the file containing the error, the line number on which the error occurs, and the error message. Clicking on the underscored error in the Building window starts a text editor that displays the source file containing the error. For more information, see "Fixing Build Errors" in the Building Programs section of the online help.

Note – Only Sun compilers produce output that can be converted to hypertext links. If you use a build command that does not call Sun compilers, you will not have links to the source files from the build errors listed in the Building window.
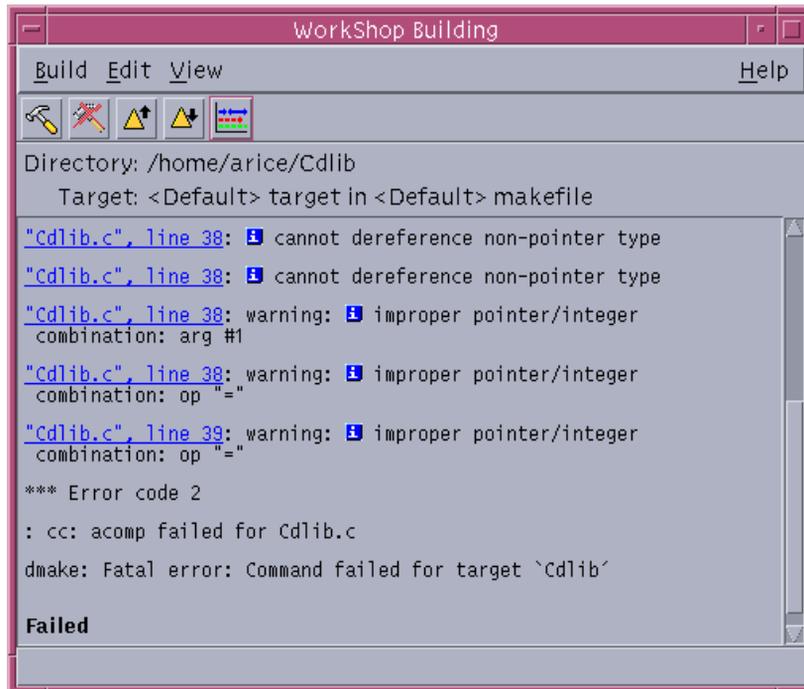
FIGURE 3-3 Build Errors in the Build Output Display Pane of the Building Window

Error messages issued by the Fortran, C, and C++ compilers include an information icon (⊞) in the build error message. Click on the icon to open a pop-up window displaying a definition of the associated error message (see FIGURE 3-4).
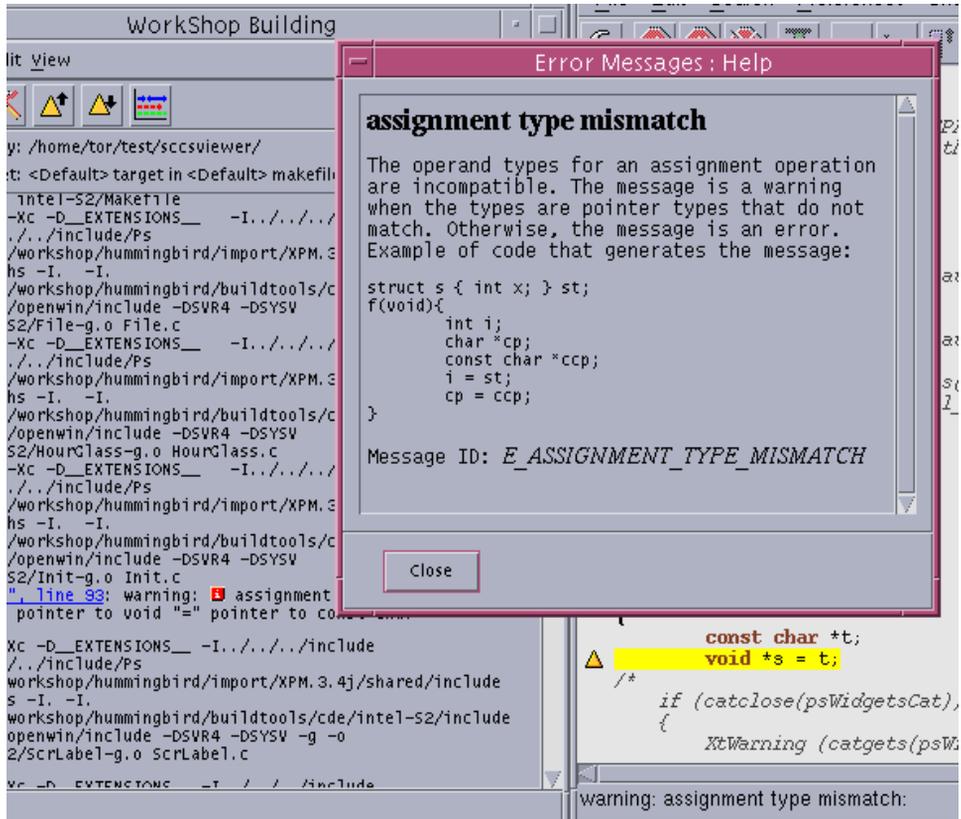
FIGURE 3-4 Build Error and Dialog Box With Associated Error Message Defined

Exiting Building

To kill the current build process and close all build windows, choose Build ➤ Exit Building in the Building window.

If you want to close the building windows without killing the current build process, choose Build ➤ Close.

Debugging a Program

Sun WorkShop provides an integrated debugging service that can run a program in a controlled fashion and inspect the state of a stopped program. Sun WorkShop gives you complete control of the dynamic execution of a program, including the collection of performance data.

Through the Debugging window (to open, choose Debug ➤ Show Debugging Window) and the windows accessed from it, you can:

- Start a debugging session or multiple debugging sessions
- Determine where your program stops executing
- Control program execution
- Use breakpoints
- Attach to processes
- Trace code
- Evaluate expressions and variables
- Use the call stack
- Fix your program
- Debug multithreaded programs
- Collect performance data
- Use runtime checking
- Graph arrays
- Set up your debugging environment
- Create custom buttons

In addition, machine-level and other commands are available to help you debug code. You can use standard dbx commands in the Dbx Commands window.

For information about debugging how-tos, concepts, windows, and dbx commands, see the Using the Debugging Window and Using dbx Commands sections of the online help (you can access the online help through the Help menu in any Sun WorkShop window).

Preparing for Debugging

To prepare for debugging, you must generate debugging information when you compile your source files by doing one of the following:

- Ask Sun WorkShop to generate debugging information when compiling your project's source files: Choose Project ➤ Edit Project in the WorkShop main window, and click Select Project Preferences. When you are finished with your selections, click OK, and then build your project. For more information, click Help in the Edit Current Project window or see "Editing a Project" in the Working With Projects section of the online help.

- Compile the application using the -g or -g0 (zero) option, which instructs the compiler to generate debugging information during compilation (for information on how to specify these options in your makefile, see Appendix B). For more detailed information on preparing your program for debugging, see *Debugging a Program with dbx.*

Starting Debugging

To start debugging a program:

1. **Choose a debugging state.**

   - Choose Debug ➤ Quick Mode to run a program normally, but with debugging ready in the background to save the program in case your program terminates abnormally. Use Quick Mode when you think you are finished debugging, want to avoid waiting for symbols to load, and want to test a fix you made.

   - Choose Debug ➤ Debug Mode or click Debug in the Sun WorkShop main window tool bar to debug the program using the full functionality of the debugging service.

2. **Select the program to debug.**

   To debug the current program, click Debug in the Sun WorkShop main window tool bar. To debug another program, do the following:

   - To debug a program previously run or debugged in Sun WorkShop, select the program from the Debug menu picklist.

   - To debug a program that is new to Sun WorkShop, load the new program by choosing Debug ➤ New Program.

   - To attach to another running process, choose Debug ➤ Attach Process.

- To debug a core dump file from an unsuccessful program execution, choose Debug ➤ Load Core File.

Your program loads, and the Debugging window (see FIGURE 4-1) and a text editor window open. Sun WorkShop allows you to view and edit a program's source code and perform basic debugging operations from a text editor window. The editor window tool bar provides access to common debugging operations, especially those that use a source component as an argument, plus buttons from other parts of Sun WorkShop.

To change your default text editor and set text editor options, see "Choosing a Text Editor and Text Editor Options" on page 20.

3. **Run your program by doing one of the following:**

- Press F6.
- Click Start in the Debugging window tool bar.
- Choose Execute ➤ Start in the Debugging window.
- Click Continue in the Debugging window tool bar.
- Choose Execute ➤ Continue in the Debugging window.

For more information about the Debugging window, see "Debugging Window" in the Using the Debugging Window section of the online help.

You can change run parameters, such as program arguments, the run directory, and environment variables, during a debugging session. For more information, see the following topics in the Using the Debugging Window section of the online help:

- "Specifying Program Arguments"
- "Specifying a Run Directory"
- "Setting Environment Variables"
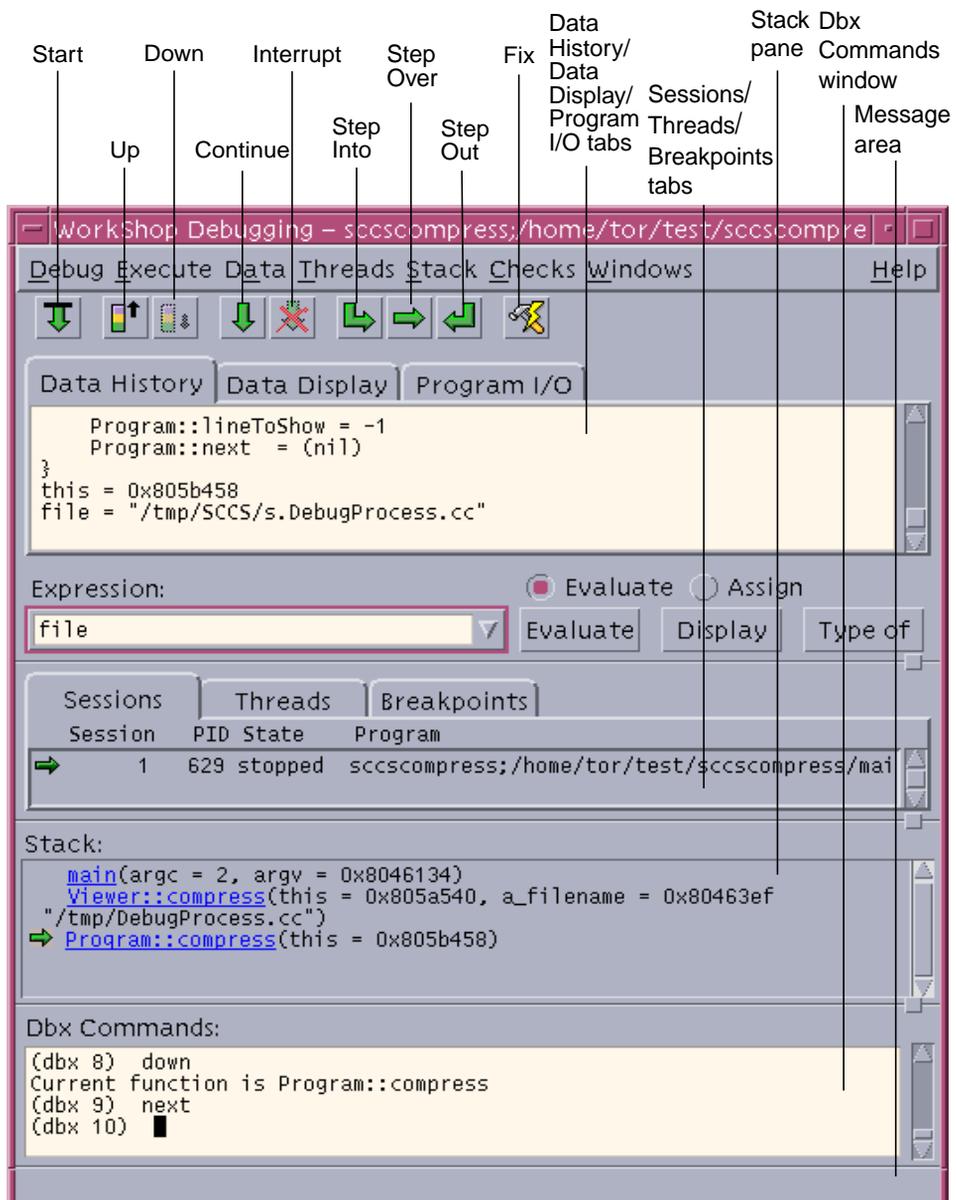
FIGURE 4-1 Debugging Window

Customizing Your Debugging Session

You can customize your debugging session to change the defaults for:

- Debugging output
- Debugging behavior
- Window layout
- Window behavior
- Data Display windows
- Language and scoping
- Runtime checking
- Data grapher
- Debugging performance
- Forks and threads
- Command-line only
- Advanced options

To view the available debugging options, choose Debug ➤ Debugging Options in the Debugging window. Using the Debugging Options dialog box, you can customize a debugging session or change defaults for the entire debugger. You can also set many of the defaults by setting dbx environment variables with the dbxenv command (for more information, see "dbxenv Command" in the Using dbx Commands section of the online help).

For detailed information about choosing debugging options, see:

- "Debugging Options Dialog Box" in the Using the Debugging Window section of the online help
- *Debugging a Program With dbx*

Stepping Through Your Code

You can view your code by stepping, which is moving through your code one line at a time. As you step, a green highlighted line known as the program counter (PC) marks your place in the program. With each step, the PC moves to the next source line to be executed (showing you the next line to be executed).

There are three ways to step:

Step Into   Proceed forward one source line. If the source line is a function call, the debugger stops before the first statement of the function.

Step Over	Proceed forward one source line. If the source line is a function call, the debugger executes the entire function without stepping through the individual function instructions.
Step Out	Finish execution of the present function and stop on the source line immediately following the call to that function. If the PC stops on the same source line as the call, there are a few more machine instructions remaining that are associated with the call. Stepping once more completes the call and you are on the next source line.

For detailed information about stepping through your code, see "Program Stepping" in the Using the Debugging Window section of the online help.

Setting Breakpoints

You can set breakpoints to force the debugger to stop execution. You can set simple breakpoints to stop at a line of code or in a procedure or function. Set advanced breakpoints to break in C++ classes, track changes in data, break on a condition, break on special events, or create your own custom breakpoints.

You can set and clear breakpoints in the editor window or the Breakpoints window. In the editor window, you can set or clear a breakpoint at a line of code or in a function. In the Breakpoints window, you can set more complex breakpoints, such as a breakpoint when a signal occurs. (The Breakpoints tab in the Debugging window displays breakpoints you have already set.)

For more information on setting and using breakpoints, see "Breakpoints" and the Using Breakpoints How-Tos in the Using the Debugging Window section of the online help.

Examining Values and Data

An evaluation is a one-time spot-check of the value of an expression. You can evaluate expressions at any time from the editor window or the Debugging window. You can track the changes in a value each time the program stops using the Data Display tab in the Debugging window or a separate Data Display window.

The results of an evaluation are listed in the Data History tab of the Debugging window. A dashed line indicates that the evaluation context has changed since the last evaluation. The Data History tab maintains a list of expressions you previously evaluated in a history list. You can clear the Data History tab at any time by choosing Data ➤ Clear History.

To evaluate an expression using the editor window, do one of the following:

- Use the balloon expression evaluator, which instantly shows you the current value of the expression at which your cursor is pointing in your editor. For more information, see "Using the Balloon Expression Evaluator" in the Text Editing section of the online help.

- Select the target variable or expression in the source display. Then do one of the following:

  - Click Evaluate or choose WorkShop ➤ Evaluate ➤ Selected to find the value of the selected expression.
  - Click Evaluate * or choose WorkShop ➤ Evaluate ➤ As Pointer to evaluate where a pointer-type expression points.

  The value is shown in the Data History tab, or when the result is short, the result is printed in the footer message area of the editor window. A separator line is inserted into the Data History tab list whenever the evaluation context changes. For more information, see "Expression Evaluation" in the Using the Debugging Window section of the online help.

Monitoring Data Values

By default, Data Display is a tab in the Debugging window. You can choose to have it shown as a separate window. For more information, see "Choosing How to Show the Data Display" in the Using the Debugging Window section of the online help.

The Data Display tab in the Debugging window allows you to watch the changes in the value of an expression during program execution. A set of expressions you choose is automatically evaluated every time a program stops executing—at a breakpoint, at a step, and when the program is interrupted. When the value of an expression changes, the value is highlighted in boldface.

The Data Display tab shows you how a value changes each time you stop execution. If you need to monitor changes in a value as the program is running, use the On Access breakpoint (see "Breaking On Access" in the Using the Debugging Window section of the online help). With this breakpoint, you can ask the program to stop whenever a specific memory location is either read or written.

From the Data Display tab or separate window, you can display pop-up windows to view additional information about an expression, giving you control over the information you are viewing.

For more information, see "Data Display Window" and "Data Display Tab" in the Using the Debugging Window section of the online help.

Collecting Performance Data

While running your program in the debugging service, you can use the Sampling Collector to collect performance data and write it to experiment files to be used by the Sampling Analyzer. The Sampling Collector can gather clock-based profiling data, thread-synchronization wait tracing data, hardware counter overflow profiling data, and address-space data. The Collector automatically records global execution statistics, including page-fault and I/O data, context switches, and working-set and paging statistics.

For more information on collecting and analyzing performance data, see:

- Chapter 6 in this book
- *Analyzing Program Performance With Sun WorkShop 6*
- The Collecting Performance Data How-Tos in the Using the Debugging Window section of the online help

Detecting Runtime Errors

Runtime checking (RTC) allows you to automatically detect runtime errors in an application during the development phase. Using RTC, you can:

- Detect memory access errors
- Detect memory leaks
- Collect data on memory use
- Work with all languages
- Work on code for which you do not have the source, such as libraries

To use runtime checking, you must turn on the type of checking you want to use before you execute the program. Then, when you run the program, RTC compiles reports on your memory usage.

For more information, see "Runtime Checking" in the Using the Debugging Window section of the online help.

Tracing Code

Tracing collects information about what is happening in your program and displays it in the Dbx Commands window. Program execution does not stop.

An unfiltered trace displays each line of source code as it is about to be executed, producing volumes of output. Filtering a trace to display information about events in your program creates more selective output. For example, you can trace each call to a function, every member function of a given name, every function in a class, or each exit from a function. You can also trace changes to a variable.

An *event* is the association of a program event with a debugging action. A typical event is a change in the value of a specified variable. A handler manages debugging events. The trace listing in the Breakpoints window is called a trace handler because it manages the trace, a type of event.

For more information, see "Code Tracing" in the Using the Debugging Window section of the online help.

Examining the Call Stack

The call stack represents all currently active routines, routines that have been called but have not yet returned to their respective caller. In the stack, the functions and their arguments are listed in the order that they were called. The initial function (main() for C and C++ programs) is at the top of the Stack pane; the function executing when the program stopped is at the bottom of the Stack pane. This function is known as the *stopped in function*.

The source code of the stopped in function is displayed in the editor window with the next line to be executed highlighted in green.

You can examine the call stack by doing any of the following:

- Move up one level in the stack by clicking Up or choosing Stack ➤ Up.
- Move down one level in the stack by clicking Down or choosing Stack ➤ Down.
- Remove multiple frames by placing your cursor next to the frame you want to return to and choosing Stack ➤ Pop to Current Frame.
- Remove the function you are stopped in from the stack by choosing Stack ➤ Pop.
- Remove multiple frames by placing your pointer next to the frame you want to return to and choosing Stack ➤ Pop to Current Frame.

Using Pop gives you a limited form of undo. If you want to start executing from the beginning of the current function again, Pop to the parent stack frame and then step into the function. You are now back at the start of the function.

For more information on using the call stack, see "The Call Stack" in the Using the Debugging Window section of the online help.

Debugging Multithreaded Programs

When a multithreaded program is detected, the Threads tab in the Debugging window opens. You can display sessions by clicking the Sessions tab. For a multithreaded program, the tab lists information about the threads in the currently selected process. The current thread is marked with a green arrow.

For more information, see "Multithreaded Program Debugging" in the Using the Debugging Window section of the online help.

Debugging Processes Simultaneously

You can debug more than one program at a time, with each program connected to a separate debugging session. Following are three examples of programs you might want to debug simultaneously:

- A process and the child process it forks
- A client and server program
- Two related programs

If you are debugging a program and you ask to debug another program, a message tells you that you are currently debugging a program. You will be prompted to terminate or detach the current session and load a new debugging session, reuse the current session, or debug both sessions. Choose to debug both only if you want to debug both programs simultaneously.

Managing Sessions

The Sessions tab in the Debugging window and the Active Sessions dialog box maintain a list of all the debugging sessions. To open the Active Sessions dialog box, choose Debug ➤ Manage Sessions in the Debugging Window. The current program is marked with an arrow.

Debugging multiple sessions consumes resources and might slow down your system. The Debugging window states how many active sessions you have. To remove sessions you no longer need, click Detach or Quit Session in the Active Sessions dialog box.

Although you are debugging multiple sessions, you can see the context of only one session at a time. When you switch to a different session, the Debugging window, the editor window, the Dbx Commands window, and the other displays change to reflect the context of the new session.

If you want to see the programs side by side (with an editor and a Debugging window for each program), you need to start two WorkShop applications. To prevent the WorkShop applications from sharing the same editor, start both WorkShop applications with the following command:

```
% workshop -s editsessionname
```

For example, start your first WorkShop application with workshop -s 1 and start your second WorkShop application with workshop -s 2. See the workshop(1) man page for more information.

For more information on managing sessions, see "Managing Sessions" in the Using the Debugging Window section of the online help.

Debugging a Child Process

When a process forks a child process, you can choose to debug the parent process, the child process, or both. You can also override the normal deletion of all breakpoints from the forked process.

For more information, see "Child Process Debugging" in the Using the Debugging Window section of the online help.

Exiting Debugging

If you want to close the Debugging window without quitting the processes under the control of Sun WorkShop, choose Debug ➤ Close. The processes under the control of Sun WorkShop continue running, using memory and CPU time. Sun WorkShop continues to store data on these processes.

To quit all processes currently under the control of Sun WorkShop and close all debugging windows, choose Debug ➤ Exit Debugging in the Debugging window.

Browsing Source Code

Sun WorkShop uses pattern search and source browsing modes in the Browsing window to browse your C, C++, Fortran 77, and Fortran 95 source code. Pattern search mode allows you to search your source code for any text string, including text embedded within comments. Source browsing mode allows you to find all occurrences of any program-defined symbol in your code by searching in a database that is created when you ask Sun WorkShop to generate source browsing information at the time you create or edit your project, compile your code with a source browsing option, or create a tags database. You then view the occurrences or matches to your query with their surrounding source code in the Browsing window match pane.

You can also graph the function and subroutine relationships in your program, and if your source code is written in C++, you can browse and graph the classes defined in your program.

For step-by-step instructions and more information about browsing, see the Browsing Source Code section of the online help (you can access the online help through the Help menu or Help button in any Sun WorkShop window).

Using Pattern Search Mode

Pattern search searches for any text string (including text embedded in your comments) in the current directory or in the directories imported in the `sb_init` file (for more information about `sb_init` and searching multiple directories, see "Searching Multiple Directories" in the Browsing Source Code section of the online help).

Use pattern search when you:

- Want to do a quick search for a text string
- Do not have a source browsing database in the directory you want to search

- Do not want to graphically view function call relationships or class hierarchies
- Do not want to examine the data or member functions of a class

Pattern search uses `grep` syntax and searches all source code lines for a match to the string you type in the Pattern text box of the Browsing window (see FIGURE 5-1). For more information, see "Searching for a Pattern" in the Browsing Source Code section of the online help.
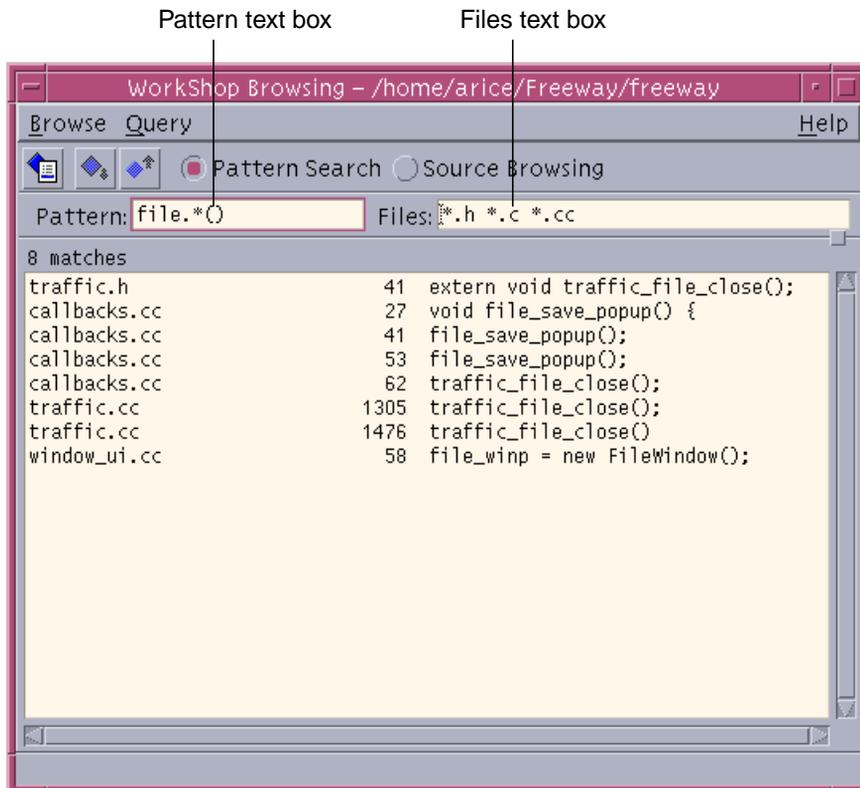
FIGURE 5-1   Browsing Window in Pattern Search Mode

Pattern Search Special Characters

Although you can type in the Pattern text box a pattern exactly as it appears in the code, you can also use special characters (wildcard characters) to specify a pattern. You can use the special characters in TABLE 5-1 in the Pattern text box in the Browsing window.

TABLE 5-1 Pattern Search Special Characters

Character	Meaning	Example
Period (.)	Matches any character.	`l.nes` matches all occurrences of `lanes` or `lines`.
Asterisk (*)	Matches zero or more occurrences of the preceding character.	`file.*()` matches any string that contains `file` followed by zero or more characters and `()`, such as `traffic_file_close()` and `file_save_popup`. `*file` matches only strings that begin with `file`.
Caret (^)	Constrains the search to match the beginning of a line.	`^tr*` finds all lines that begin with `traffic`, `truck`, or any other string beginning with `tr`.
Dollar sign ($)	Constrains the search to match the end of a line.	`lanes$` finds all the lines that end with the string `lanes`.
Backslash left angle bracket (\<)	Matches the start of a word.	`\<get` finds `get_foobar`, but not `widget`.
Backslash right angle bracket (\>)	Matches the end of a word.	`\<String\>` finds `String *foo`, but not `XmStringCreate()`.

Surrounding an expression with a caret and a dollar sign constrains the search to match the entire line.

For more information, see "Special Characters in Pattern Search and Source Browsing Modes" in the Browsing Source Code section of the online help.

Multiple Directory Searches

Pattern searching uses the directories listed in the `sb_init` text file to search source files in multiple directories. For step-by-step instructions, see "Searching Multiple Directories" in the Browsing Source Code section of the online help.

Using Source Browsing Mode

In source browsing mode, Sun WorkShop responds to queries by searching in a database that contains information about the source files you are browsing. Use source browsing mode when you:

■ Have a source browsing database

■ Want to search for language elements such as functions, classes, structs, unions, or records or their usage, definitions, or assignments

■ Want to graphically view function call relationships or class hierarchies

■ Want to examine the data or member functions of a class

FIGURE 5-2 shows the Browsing window in Source Browsing mode.
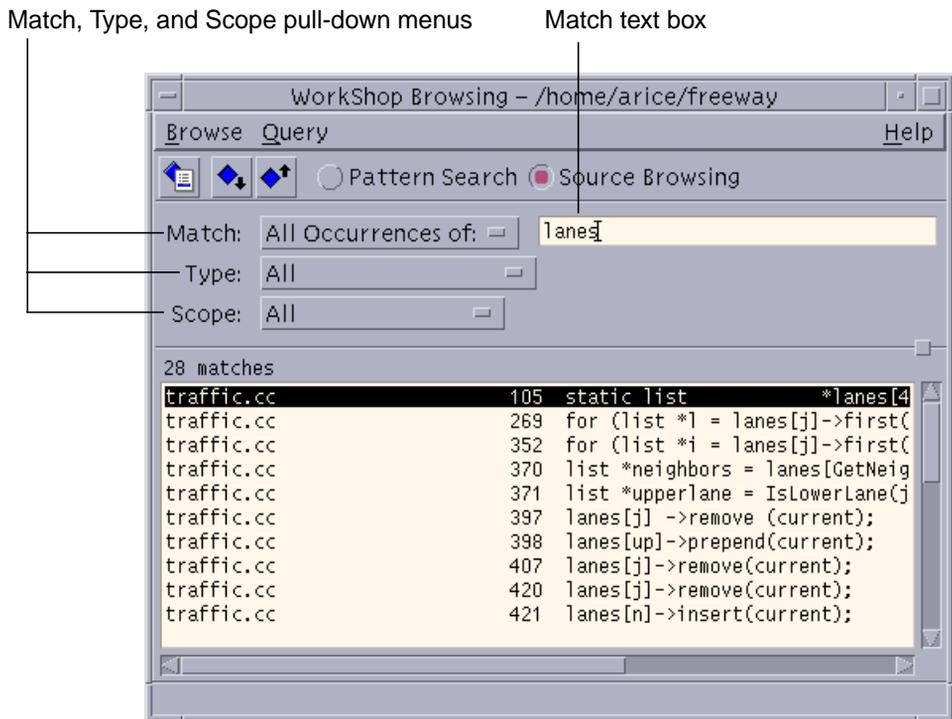
FIGURE 5-2   Browsing Window in Source Browsing Mode

Source Browsing Databases

Sun WorkShop obtains its browsing information from a database that describes the static structure of your program. How the browser functions depends upon the database it accesses. The following are the browser database choices available:

- **No database.** You must use pattern search mode instead of source browsing mode. For more information, see "Searching for a Pattern" in the Browsing Source Code section of the online help.

- **Compiler-generated browser database.** This database provides full browser functionality. Source browsing mode responds to queries by searching through this database.

  One of the selections you can make when you are creating or editing a project is for Sun WorkShop to generate the database when it compiles your code or you can generate the database by adding the appropriate source browser option to your makefile and building your source files. For step-by-step instructions, see "Generating a Browser Database" in the Browsing Source Code section of the online help.

- **Tags-generated database.** This database provides a way to browse source files without compilation, allows queries on functions and global variables, and displays function calls (graphing features not available). A tags database recognizes only global definitions for variables, types, and functions and collects information on function calls. Function calls for C++ members are recognized only when members are called explicitly.

  For step-by-step instructions, see "Creating a Tags Database" in the Browsing Source Code section of the online help.

Source Browsing Special Characters

Although you can type a name or function in the Match text box exactly as it appears in your source code, you can also use special characters (wildcard characters) to specify a set of character strings.

You can use the special characters in TABLE 5-2 in the Match text box in the Browsing window.

TABLE 5-2 Source Browsing Special Characters

Character	Meaning	Example
Period (.)	Matches any character.	`.ehicle` matches all occurrences of `vehicle` or `Vehicle`.
Asterisk (*)	Matches zero or more occurrences of the preceding character.	`vehi.*` matches any string that begins with `veh`, such as `vehicle_length().vehi*` matches `veh.`, but not `vehicle_length()`.

For more information, see "Special Characters in Pattern Search and Source Browsing Modes" in the Browsing Source Code section of the online help.

Multiple Directory Browsing

For all projects except a user makefile project: When you create or edit your project, if you ask Sun WorkShop to generate source browsing information during compilation, Sun WorkShop merges all the browsing directories for you. Then when you search in pattern search mode or source browsing mode, all the source directories in your project are automatically searched. For more information, see "Creating a New Project" and "Editing a Project" in the Working With Projects section of the online help.

For a user makefile project: If you keep your source files in several directories, you will most likely run the compiler in each of these directories. By default, the compiler generates a separate source browsing database in each directory. Since Sun WorkShop browses only one database at a time, it searches only that part of your application located in the current directory. You can override this default behavior by importing databases. For step-by-step instructions for how to import databases, see "Browsing Multiple Directories" in the Browsing Source Code section of the online help.

Relating Browsing and Graphing

FIGURE 5-3 shows how the Browsing window, the Call Graph window, the Class Graph window, and the Class Browser window interrelate.
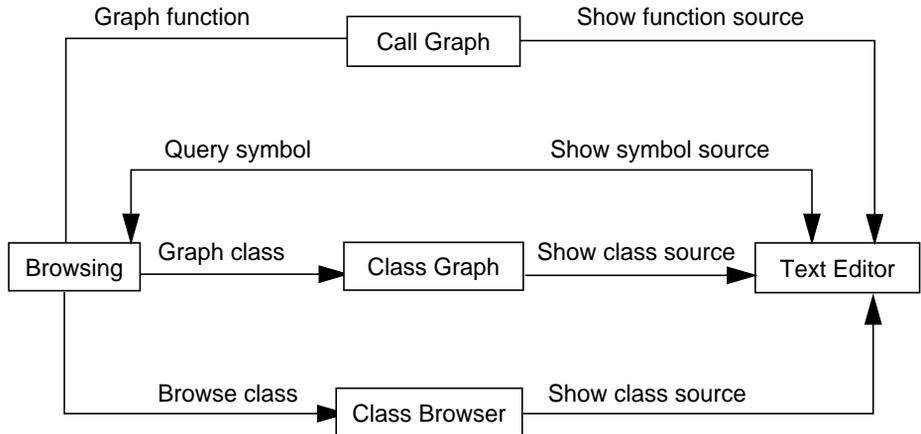
FIGURE 5-3   How Browsing, the Graphers, and the Class Browser Interrelate

Graphing Functions

Using the Call Graph window, you can graphically inspect the relationships of the functions in programs using ANSI C, C++, and Fortran. You can display the functions that either call or are called by one or more selected functions. The Call Graph window provides a graphic representation of the call relationship of functions and subroutines. For step-by-step instructions and more information, see "Graphing a Function Call" in the Browsing Source Code section of the online help.

You must have a source browsing database to view function relationships (see "Source Browsing Databases" on page 47).

Note – You can graph virtual functions, but Sun WorkShop cannot determine the actual function that would be called. For example, if `main` calls `b::d()`, a virtual function that could actually call `b1::d()` or `b2::d()`, Sun WorkShop cannot tell which function is called. The graph shows `main` calling `b::d()`, but no connection between `main` and `b1::d()` or `main` and `b2::d()`.

To change the colors used for node background, graph pane background, node border, node text, and arrows between nodes in the Call Graph window, edit the WORKSHOP resource file (see "Call Graph and Class Graph Window Colors" on page 70). Any color changes you make apply to both the Call Graph and Class Graph windows.

FIGURE 5-4 shows the Call Graph window.
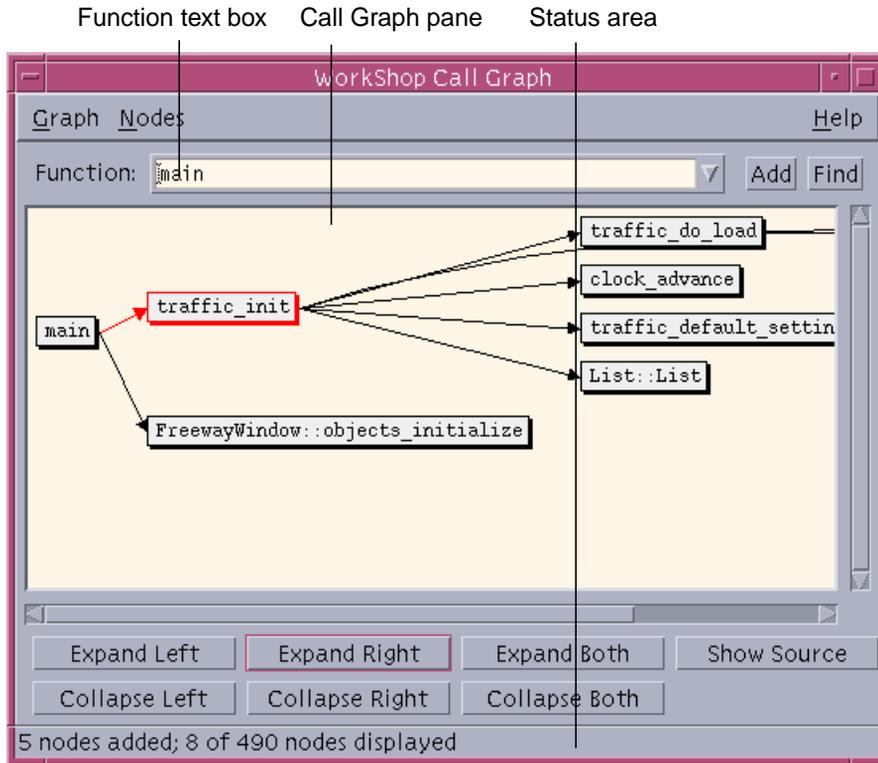
FIGURE 5-4   Call Graph Window

Graphing Classes

Using the Class Graph window, you can graphically inspect the inheritance structure of classes in C++ programs. The Class Graph window provides a graphic representation of class hierarchies. For step-by-step instructions and more information, see "Graphing a Class Hierarchy" in the Browsing Source Code section of the online help.

To change the colors used for node background, graph pane background, node border, node text, and arrows between nodes in the Class Graph window, edit the WORKSHOP resource file (see "Call Graph and Class Graph Window Colors" on page 70). Any color changes you make apply to both the Class Graph and Call Graph windows.

FIGURE 5-5 shows the Class Graph window.

Class text box   Message footer
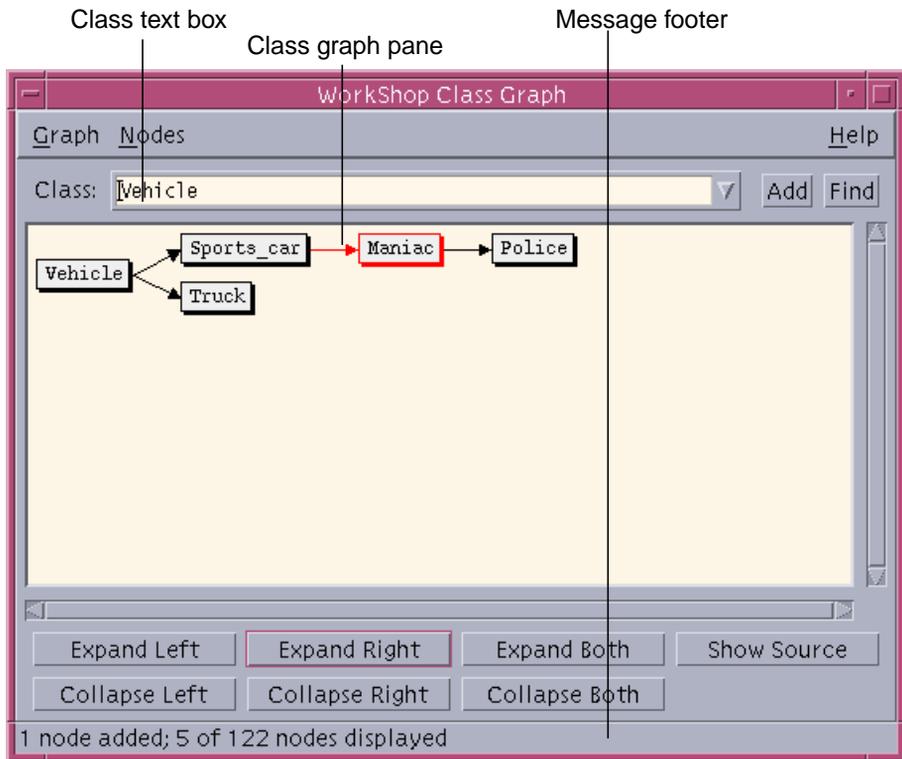
                    Class graph pane

FIGURE 5-5   Class Graph Window

Browsing Classes

Using the Class Browser, you can:

- **Browse a class.** You can show the class list and data function members and view class interfaces and relationships.
- **Examine class relationships.** You can select a class and examine its base, derived, and friend classes, and you can browse classes, structs, and unions referenced in the current class.
- **Graph a class.** You can graph the class hierarchy of a class selected in the Class Browser window.
- **Show the source of a class.** You can show the source of a particular class in an editor window.

You can view information about classes and their member and friend functions in the Class Browser window. By navigating through the classes in the source code and libraries, you can understand how the classes were defined and used.

When you open the Class Browser window (see FIGURE 5-6), the Browser list contains all classes of the type Class or Struct in the current source browser database. Using the two check boxes to the right of the Browser list, you can show all types, only classes and structs, or only the unions.

For step-by-step instructions and more information, see "Browsing a Class" in the Browsing Source Code section of the online help.
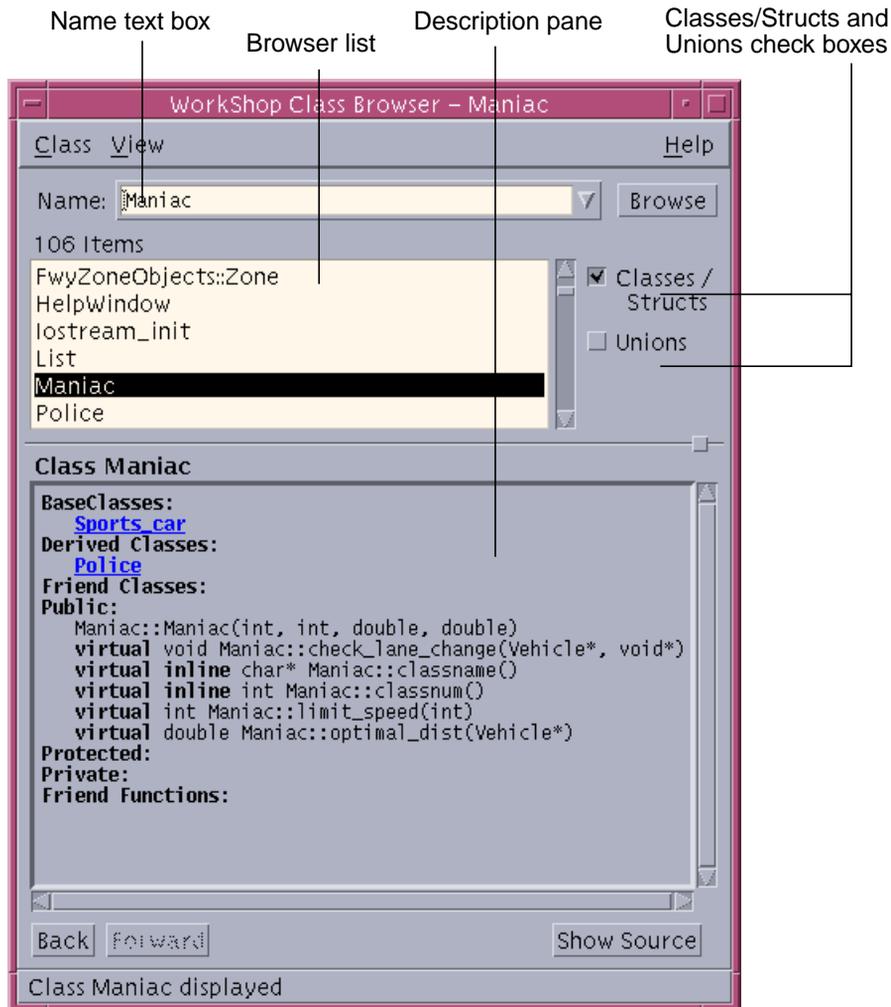
FIGURE 5-6 Class Browser Window

Exiting Browsing

To quit the current browsing process and close all browsing windows, choose Browse ➤ Exit Browsing in the Browsing window. If you want to close the Browsing windows without killing the current browse process, choose Browse ➤ Close.

CHAPTER **6**

Analyzing Program Performance

This chapter describes the basic features of the Sun WorkShop Sampling Collector and Sampling Analyzer. The UNIX `prof` and `gprof` performance-profiling tools generate only user CPU information. With the Sampling Collector and Sampling Analyzer, you can examine a wider range of metrics, broken down by functions, load objects, sampling intervals, or threads and lightweight processes (LWPs) in multithreaded programs:

- The Sampling Collector gathers performance data during the execution of an application and saves it to an experiment-record file. Start the Sampling Collector from the Windows menu in the Debugging window,

- The Sampling Analyzer displays the performance data in the experiment-record file so you can analyze your program's performance and determine where it can be improved. Start the Sampling Analyzer from the Tools menu in the Sun WorkShop main window,

For detailed information about how to use these and other performance-profiling tools included in Sun WorkShop, see:

- *Analyzing Program Performance With Sun WorkShop 6*
- The Analyzing Program Performance section of the Sun WorkShop online help
- The links under "Collecting Performance Data" in the Using the Debugging Window section of the Sun WorkShop online help

---

Note – Before collecting performance data you must build your application. For information on building, see Chapter 3 and the Building Programs section of the Sun WorkShop online help (you can access the online help through the Help menu in any Sun WorkShop window).

Collecting Performance Data

The Sampling Collector gathers performance data about a program as it runs in the Debugging window. It stores the data in an experiment-record file, which you then load into the Sampling Analyzer.

TABLE 6-1 describes the types of data you can collect.

TABLE 6-1   Types of Data to Collect

Type	Description
Clock-based profiling data	Function or load-object timing information.
Thread-synchronization wait tracing data	Wait time on calls to thread-synchronization routines in multithreaded programs.
Address-space data	Information about how your application uses the pages and segments in its address space

The Collector automatically records global execution statistics, including page-fault and I/O data, context switches, and working-set and paging statistics. In the Collector you can also specify the length of sampling intervals during which data is collected, or choose to terminate each sampling interval manually.

For detailed information about choosing the types of data to collect and running the Collector, see:

- *Analyzing Program Performance With Sun WorkShop 6*
- "Choosing the Data to Collect" in the Sun WorkShop online help

---

Note – You can also run the Sampling Collector from dbx. See *Analyzing Program Performance With Sun WorkShop 6* for more information.

Analyzing Performance Data

After you collect performance data with the Sampling Collector, you can view it in the Sampling Analyzer, a separate tool that you start from the Tools menu in the Sun WorkShop main window or the Sampling Collector window. The Analyzer's various displays pinpoint where your program is spending excessive execution time or otherwise making inefficient use of system resources.

The Sampling Analyzer window gives you a choice of four displays that you can choose from the Data list box (see TABLE 6-2).

TABLE 6-2 Types of Data to View and Analyze

Type	Description
Function List display	Shows detailed information about your program's functions and load objects.
Overview display	Shows microstate accounting information for program sampling intervals.
Address Space display	Shows use of pages or sectors in your program's address space.
Execution Statistics display	Shows global statistics over the program execution time.

In the Analyzer displays, you can examine data for your whole program, or you can specify individual functions, load objects, sampling intervals, threads, and LWPs for analysis. You can also use the Analyzer to generate a mapfile that the linker can use to make your program more efficient in its use of the address space.

The Analyzer allows you to look at Function List data for more than one experiment record at a time. When you have more than one experiment loaded, data-filtering capabilities are disabled, as are the other displays.

For detailed information about how to use the Sampling Analyzer, see:

- *Analyzing Program Performance With Sun WorkShop 6*
- The Analyzing Program Performance section of the Sun WorkShop online help

Examining Function and Load-Object Metrics

The Function List display shows for each function or load object the exclusive and inclusive values for the following metrics:

- Clock-based profiling, including user CPU time, total LWP time, wall-clock time, and system and page-fault times
- Thread-synchronization wait tracing for multithreaded programs

Each metric is displayed as an absolute value (seconds, counts) and as a percentage of the total program metric.

You can specify which of the available metrics you want to appear in the Function List display and the metric upon which the data is sorted. You can also open a window that lists all available metrics for a selected function or load object.

Examining Caller and Callee Metrics

From the Function List display, you can start the Callers-Callees window, where you can examine exclusive, inclusive, and attributed data for a selected function, its callers, and its callees. You can also step through the program structure by selecting callers or callees of the selected function, which then becomes the selected function. For more information, access the online help through the Help menu or the Help button in any window.

Displaying Annotated Source and Disassembly Code

To examine program performance line by line or instruction by instruction, you can display source code and disassembly code annotated with program metrics. These metrics enable you to pinpoint within a given function which line or lines are using up the most execution time. For more information, access the online help through the Help menu in any window.

Merging Source Files

Merging lets you compare two text files, merge two files into a single new file, and compare two edited versions of a file against the original to create a new file that contains all new edits. Merging loads and displays two text files for side-by-side comparison, each in a read-only text pane. Any difference between the two files is marked. A merged version of the two files, which you can edit to produce a final merged version, is displayed.

When you load the two files to be merged, you can also specify a third file from which the two files were created. When you have specified this ancestor file, Merging marks lines in the descendants that are different from the ancestor and produces a merged file based on all three files.

For more information, see the online help (choose Help ➤ Contents in the Merging window to access online help).

Loading Files into Merging

Load files into merging by following these instructions:

1. **Choose Tools ➤ Merging from the WorkShop main window.**

   The Merging window opens (see FIGURE 7-1). The Merging window is divided into three panes: two side-by-side panes, which display different versions of the file, and the merged result in the bottom pane. The top two panes are read-only, the bottom pane contains selected lines from either or both versions of the file and can be edited to produce a final merged version.

2. **Choose File ➤ Open.**

3. **In the Directory text box, select a working directory.**

   This is the default directory used to select and save files. The browse button to the right of the text box displays a dialog box in which you can select a directory.

4. **In the Left File and Right File text boxes, select the two files you want to compare.**

5. **If you are comparing the files against a common ancestor, type the earlier version of the two files in the Ancestor File text box.**

   An ancestor file is required to use Auto Merge.

6. **If you want to specify the name of the output file, type it in the Output File text box.**

   The name `filemerge.out` is the default, and the file is stored in the working directory.

7. **Click Open to load the files.**

   The names of the left file, right file, and output file are displayed above each text pane. In a three-way comparison, the name of the ancestor file is displayed in the window header.
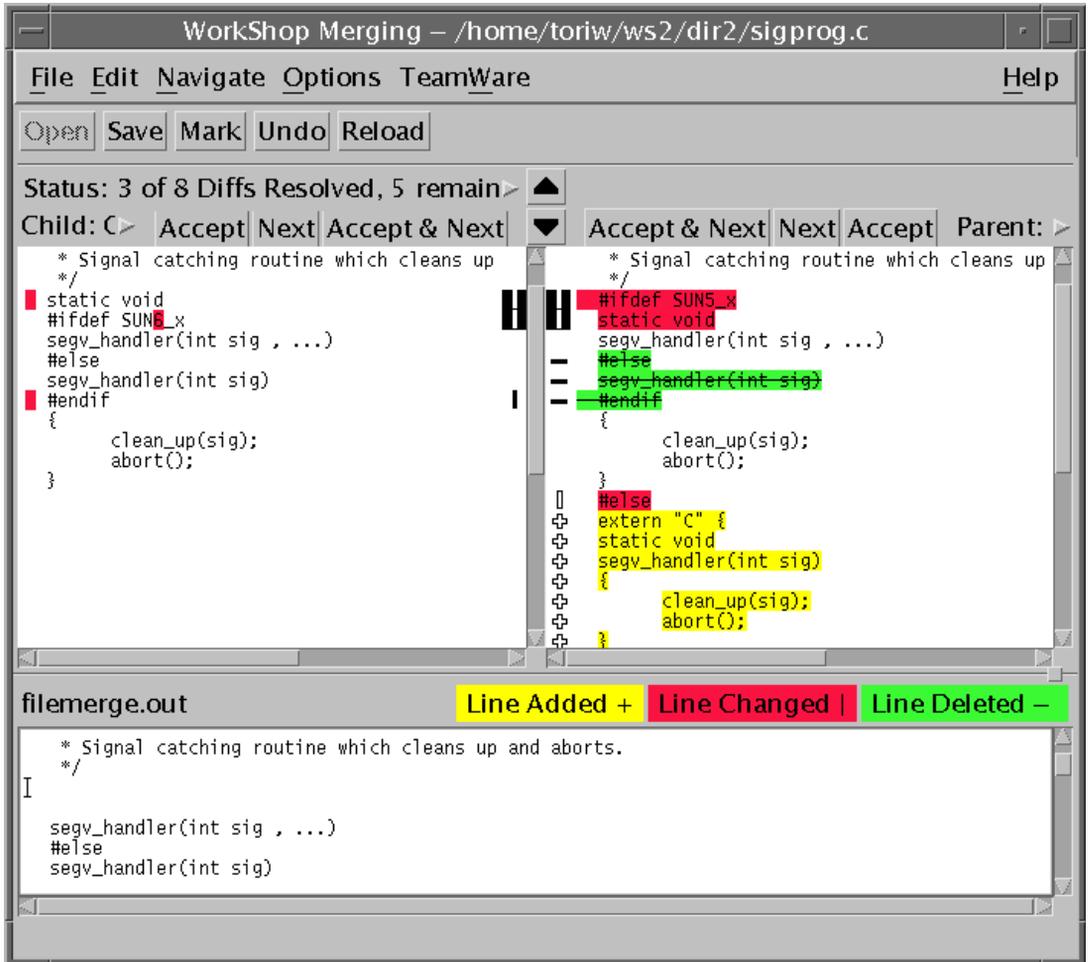
FIGURE 7-1   Merging Window

Working With Differences

Merging operates on differences between files. When merging discovers a line that differs between the two files to be merged (or between either of the two files and an ancestor), it marks the lines in the two files with icons corresponding to how the lines differ. Together, these marked lines are called a difference. As you move through the files from one difference to the next, the lines that differ and their icons are highlighted.

The highlighted difference is called the current difference. The differences immediately before and immediately after are called the previous difference and the next difference. A difference is resolved if the changes to a line are accepted. A remaining difference is one that has not yet been resolved.

Reading Merging Icons

To help you find differences more easily, Merging highlights lines that differ with color and icons. Yellow shows an addition, red shows a change, green shows a deletion.

The meaning of icons is different if you are comparing two versions with each other (two input files), or if you identify an ancestor for the two versions of the file (three input files).

Two Input Files

When only two files have been loaded into Merging, lines in each file are marked by icons to indicate when they differ from corresponding lines in the other file:

- If two lines are identical, no icon is displayed.
- If two lines are different, a vertical bar (|) is displayed next to the line in each input text pane, and the different characters are highlighted in yellow.
- If a line appears in one file but not in the other, a plus sign (+) is displayed next to the line in the file where it appears, and the different characters are highlighted in red.
- Resolved differences are marked by icons in outline font.

Three Input Files

When you load two files to be merged, you can also specify a third file, called the ancestor of the two files. An ancestor file is any earlier version of the two files. When you identify an ancestor file, it is used as a basis to compare the two files and automatic merging can be done. Merging marks all lines in the derived files or their descendants that differ from the ancestor and produces a merged file based on all three files.

The lines in the files that are different from the ancestor file are marked with change bars and colors. Here's what each means:

- If a line is identical in all three files, no icon is displayed.

- If a line is not in the ancestor but was added to one or both of the descendants, a plus sign (+) is displayed next to the line in the file where the line was added, and the different characters are highlighted in yellow.
- If a line is in the ancestor but has been changed in one or both of the descendants, a vertical bar (|) is displayed next to the line in the file where the line was changed, and the different characters are highlighted in red.
- If a line is present in the ancestor but was removed from one or both of the descendants, a minus sign (-) is displayed next to the line in the file from which the line was removed, and the different characters are highlighted in green and in strikethrough.
- Resolved differences are marked by icons in outline font.

Moving Between Differences

You can move between differences using the buttons above the two panes or through the Navigate menu. Use the Previous and Next buttons to scroll through the differences without accepting them. Choose Navigate ➤ Find to navigate to a particular text string. Choose Navigate ➤ Goto Line to navigate by line numbers.

You can also navigate between differences by using the popup menu that is available in the Child and Parent panes. Click the right mouse button in either pane to open the menu.

Resolving Differences

Accept the change in either the left or right pane to resolve a difference. To accept a difference, do one of the following:

- Click the Accept button to accept the difference.
- Click the Accept & Next button to accept the difference and move to the next difference.

For more information, see the online help (choose Help ➤ Contents in the Merging window to access online help).

Setting Difference Options

Choose Options ➤ Diff Options to customize merging to ignore certain kinds of differences between files. You can set merging to ignore trailing or embedded white space and to ignore differences in case.

For more information, see the online help (choose Help ➤ Contents in the Merging window to access online help).

Merging Automatically

Merging can resolve differences automatically, based on the following rules:

- If a line has not changed in all three files, it is placed in the output file.
- If a line has changed in one of the descendants, the changed line is placed in the output file. A change could be the addition or removal of an entire line or an alteration to some part of a line.
- If identical changes have been made to a line in both descendants, the changed line is placed in the output file.
- If a line has been edited in both descendant files so that it is different in all three files, no line is placed in the output file. You must decide how to resolve the difference, by either choosing a line from a descendant or by editing the merged file by hand.

When merging automatically resolves a difference, it changes the icons to outline font. Merging lets you examine automatically resolved differences to be sure that it has made the correct choices.

You can disable Auto Merge by choosing Options ➤ Auto Merge. When automatic merging is disabled, the output file contains only the lines that are identical in all three files. You must then resolve the differences.

If you do not specify an ancestor file, merging has no reference to which to compare a difference between the two input files. Consequently, merging cannot determine which line in a difference is likely to represent the desired change. The result of an auto merge with no ancestor is the same as disabling automatic merging: Merging constructs a merged file using only lines that are identical in both input files. You must resolve the differences.

Saving the Output File

Save the output file by clicking the Save button or choosing File ➤ Save. The name of the output file is the name you specify in the Output File text box.

To change the name of the output file while saving, choose Save As and fill in the new file and directory names in the Save As dialog window.

Setting Merging Options

Use the Options menu in the Merging window to set various merging options. The menu items enable you to:

- Create a merged version of the files automatically
- Control whether files scroll separately or by corresponding lines
- Control if line numbers or line ends are displayed
- Customize tab stops
- Set how white space and case differences are handled

For more information, see the online help (choose Help ➤ Contents in the Merging window to access online help).

Sun WorkShop and Text Editor Resources

This appendix describes the resources that you can set and gives you the information you need to change the settings.

Changes to Resource Settings

Sun WorkShop uses two resource files:

- `WORKSHOP` contains the resource settings for Sun WorkShop windows, including the Browsing and Debugging windows.
- `ESERVE` contains text editor resource settings.

Each resource file has two variations: one for CDE (Common Desktop Environment) and one for non-CDE environments. The CDE version does not define generalized color and font resources for Motif elements; it allows the CDE Style Manager to control these elements.

Both the `WORKSHOP` and `ESERVE` files contain comments that indicate what a group of resources pertains to. For example, the following group of resources controls the colors used in the text editors for highlighting:

```
! Resources for highlight colors used by WORKSHOP in the editors

WORKSHOP.curPCColor:    #8BD98B
WORKSHOP.visitPCColor: #EDC9FF
WORKSHOP.breakptColor: #FF9696
```

See the `workshop(1)` man page for more information about the resource files.

To change the default value of a resource, do the following:

1. **Depending upon the resources you want to change, create a file called** `WORKSHOP` **or** `ESERVE` **in your home directory (or the directory specified in your** `XFILESEARCHPATH` **or** `XAPPLRESDIR` **environment variable).**

2. **Go to the directory where the installed resource file is located.**

   The resource files are located in the directory where Sun WorkShop is installed:

   `/opt/SUNWspro/WS6/lib/locale/`*lang*`/app-defaults/CDE`

   `/opt/SunWspro/WS6/lib/locale/`*lang*`/app-defaults/non-CDE`

   lang is your current locale (for example, `C` or `ja`).

   Note – If your Sun WorkShop software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.
   ---

3. **In the installed** `WORKSHOP` **or** `ESERVE` **resource file, copy the resources and default values that you want to change.**

4. **Paste the resources and default values into the file you created in your home directory.**

5. **Change the resource values per the instructions in this appendix.**

6. **Save the file.**

7. **Start (or exit and restart) Sun WorkShop.**

Editable Sun WorkShop Resources

TABLE A-1 through TABLE A-17 list the Sun WorkShop resources that you can change in the `WORKSHOP` resource file.

Resources that affect components in the core Sun WorkShop product do not affect Sun WorkShop TeamWare components or any component started from the Tools menu in the WorkShop main window.

Note – If you modify the default colors in Sun WorkShop to use a non-specified color, you might cause Sun WorkShop to fill up the color map.

Highlight Colors in Editor Windows

The resources listed in TABLE A-1 control the colors used to highlight functions, breakpoints, query matches, and build errors in source code displayed in the text editor windows (for an example of highlighting, see FIGURE 4-1).

TABLE A-1   Editor Highlight Color Resources

Resource Name	Description	Default Value
WORKSHOP.curPCColor	Current function	#8BD98B
WORKSHOP.visitPCColor	Visited function	#EDC9FF
WORKSHOP.breakptColor	Breakpoint	#FF9696
WORKSHOP.disabledBreakptColor	Disabled breakpoint	#BDBDBD
WORKSHOP.matchColor	Pattern or symbol match	#99CFFF
WORKSHOP.errorColor	Current build error	#FFCC40

Data Graph Window Colors

The resources listed in TABLE A-2 control the colors used in the graph types in the Data Graph window of the debugger (see *Debugging a Program With dbx*).

TABLE A-2   Data Graph Window Color Resources

Resource Name	Description	Default Value
WORKSHOP.dgLineColor	Color for Line graph type	#OOOOFF
WORKSHOP.dgFillColor	Color for Fill graph type	#FDF5E6
WORKSHOP.dgMeshColor	Color for Mesh graph type	#OOOOFF

Call Graph and Class Graph Window Colors

The resources listed in TABLE A-3 control the colors of the nodes, the lines (or arrows) connecting the nodes, and background color of the graph pane in the Call Graph window (see FIGURE 5-4) and the Class Graph window (see FIGURE 5-5).

TABLE A-3 Class Graph and Call Graph Window Resources

Resource Name	Description	Default Value
WORKSHOP*labelNodeBackground	Background color of each node	#EFEFEF
WORKSHOP*viewBackground	Graph pane background (Default uses X's Old Lace)	#FDF5E6
Node Properties When Unhighlighted		
WORKSHOP*arcForeground	Arrow between nodes	#000000
WORKSHOP*nodeForegroundColor	Node border	#000000
WORKSHOP*labelNodeForeground	Node text	#000000
Node Properties When Highlighted		
WORKSHOP*arcHighlightColor	Arrow between nodes	#FF0000
WORKSHOP*nodeHighlightColor	Node border	#FF0000

Audible Warnings

The resource listed in TABLE A-4 enables you to turn on and turn off audible warning beeps. The possible values are –XmBell and –XmNONE.

TABLE A-4 Audible Warning Resources

Resource Name	Description	Default Value
WORKSHOP*audibleWarning	Turns audible beeps on and off	XmBell

Debugger Buttons

The resource listed in TABLE A-5 enables you to set the delay in milliseconds before debugger and text editor buttons are disabled when dbx starts. This disabling prevents button flashes when you are stepping through code. If you are running Sun WorkShop on a slow system or over an ISDN line, you might want to increase this delay.

TABLE A-5   Debugger Button Disable Delay Resource

Resource Name	Description	Default Value
WORKSHOP.ButtonDisableDelay	Delays disabling of debugger and text editor buttons when dbx starts	250

Dbx Commands and Program I/O Window Output Lines

The resource listed in TABLE A-6 sets the number of lines of output to save in the Dbx Commands window and the Program Input/Output window.

TABLE A-6   Dbx Commands and Program I/O Windows Output Line Resource

Resource	Default Value
WORKSHOP*dtTerm.saveLines	1000

Project make Command

The resource listed in TABLE A-7 sets the make command used to build projects. It must accept the -f flag as well as the target and macros operands. For example, you can write your own wrapper around make that filters out certain warnings and passes the flags on to make. See the dmake(1) and make(1) man pages for more information.

TABLE A-7   Project make Command Resource

Resource	Default Value
WORKSHOP.ProjectMakeCommand	dmake -m serial

Browser Used to Display Web Updates

The resource listed in TABLE A-8 enables you to change the default path for the browser used to display the Sun WorkShop Web updates page (to access the Web updates page, choose Help ➤ Web Updates from any Sun WorkShop window).

TABLE A-8   Web Updates Browser Resource

Resource	Description	Default Value
WORKSHOP.browser	Path to browser used to display Web updates	sdtwebclient

Character Fonts in Hyperlink Windows

Many of the windows in Sun WorkShop use hyperlinks to connect to other windows to facilitate the display of related information. For example, clicking on a build error in the Building window causes an editor window to display the source code file that contains the error. Certain resources serve as flags indicating that non-ASCII characters written to a hyperlink display are to be interpreted as multibyte characters. The multibyte characters are displayed in the font indicated by the resource. The resources should be set *only* in locales in which there is to be a multibyte interpretation of non-ASCII characters.

The names of the resources as they would appear if set in the WORKSHOP resource file are:

```
WORKSHOP*HTML*WCfont:
WORKSHOP*HTML*boldWCFont:
WORKSHOP*HTML*plainWCFont:
WORKSHOP*HTML*plainboldWCFont:
WORKSHOP*HTML*Font:
WORKSHOP*HTML*boldFont:
WORKSHOP*HTML*plainFont:
WORKSHOP*HTML*plainboldFont:
```

Each WC (wide-character) font resource corresponds to a non-WC font resource. If the WC font resource is set, WC font dimensions determine the line spacing and baseline of text elements written in both the WC font and corresponding non-WC font. The purpose is to produce consistent spacing of a line where ASCII and multibyte characters are mixed. The WC font dimensions are also used for formatting a line written only in the non-WC fonts.

Where `WC` font resources are set for hyperlink displays of multibyte characters and you change a `WC` font resource, the size and spacing of `WC` fonts should be proportional to the size and spacing of non-`WC` fonts. To get proportional formatting you might need to modify the resources for non-`WC` fonts.

Hyperlink Resources

The resources listed in TABLE A-9 set the font type, weight, and angle used in hyperlinks in the windows and dialog boxes of the English version of Sun WorkShop. For examples of hyperlinks in Sun WorkShop windows, see FIGURE 3-3, which shows build error links in the Building window.

TABLE A-9   English (`C`) Locale Hyperlink Font Resources

Resource Name	Default Value
WORKSHOP*HTML*BoldFont	-*-lucida-bold-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*HTML*PlainFont	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*HTML*PlainboldFont	-*-lucidatypewriter-bold-r-normal-*-12-*-*-*-*-*-iso8859-1

TABLE A-10 lists hyperlink wide-character (`WC`) font resources for locales with multi-byte characters. If set, non-ASCII characters written to HTML displays are interpreted as multibyte characters and displayed with font indicated by the resource.

TABLE A-10   Japanese (`ja`) Locale Hyperlink Font Resources

Resource Name	Default Value
WORKSHOP*HTML*boldWCFont	-jis-fixed-medium-r-normal--16-150-75-75-c-160-*-0
WORKSHOP*HTML*plainWCFont	-jis-fixed-medium-r-normal--16-150-75-75-c-160-*-0
WORKSHOP*HTML*plainboldWCFont	-jis-fixed-medium-r-normal--16-150-75-75-c-160-*-0

Automatic Text Wrapping

The resource listed in TABLE A-11 lets you set text to automatically wrap or start a new line in a Sun WorkShop window. The default value is `True`, which means that text automatically wraps when it meets a window border.

TABLE A-11   Automatic Text Wrapping Resource

Resource Name	Default Value
`WORKSHOP*HTML*wrapPreformatText`	`True`

Vertical Scrollbars

The resource listed in TABLE A-12 enables you to turn vertical scrollbars off or on.

TABLE A-12   Vertical Scrollbar Resource

Resource Name	Default Value
`WORKSHOP*HTML*verticalScrollbarAlways`	`True`

Motif-Specific Resources

TABLE A-13 through TABLE A-17 list resources that are specific to Motif environments only and are not used by CDE.

TABLE A-13   Motif (non-CDE) Windowing Systems Font Resources

Resource Name	Description	Default Value
`WORKSHOP.labelFontList`	Label font	`-*-lucida-medium-r-normal-*-12-*-*-*-*-*-*-*`
`WORKSHOP.buttonFontList`	Button font	`-*-lucida-medium-r-normal-*-12-*-*-*-*-*-*-*`
`WORKSHOP.textFontList`	List font	`-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-*-*`

In your resource file, uncomment the resources listed in TABLE A-14 to change the fonts in a specific Sun WorkShop window.

TABLE A-14   Window Font Resources

Resource Name	Default Value
WORKSHOP*ipeDbxCommandWindow*userFont	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*ipeProgramIOShell*userFont	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*threadsList*fontList	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*handlerList*fontList	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1
WORKSHOP*processList*fontList	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-iso8859-1

This resource listed in TABLE A-15 is applicable to text in a tabular format, such as tables.

TABLE A-15   Tabular Windows Font Resource

Resource Name	Default Value
WORKSHOP.DataMonospacedFont	-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-*-*

Window Foreground and Background Colors

TABLE A-16 lists the resources that control the foreground and background colors used in most Sun WorkShop windows.

TABLE A-16   Windows, Dialog Boxes, Menus, and Buttons Color Resources

Resource Name	Description	Default Value
WORKSHOP*foreground	Foreground color of windows (text such as labels)	#000000
WORKSHOP*XmTextField*background	Background color of text boxes	#FFFFFF
WORKSHOP*XmText*background	Text color	#FFFFFF
WORKSHOP*threadsList.background	Background color of Threads pane	#FFFFFF
WORKSHOP*ipeDbxCommandWindow*dtTerm.background	Background color of Dbx Commands window	#FFFFFF
WORKSHOP*ipeProgramIOShell*dtTerm.background	Background color of Program Input/Output window	#FFFFFF

Resource Name	Description	Default Value
WORKSHOP*XmDrawingArea.background	Background color of Stack pane, Data Display, and so forth	#FFFFFF
WORKSHOP*background	Background color of Sun WorkShop windows	#DEDEDE
WORKSHOP*XmPushButton*background	Background color of buttons	#DEDEDE
WORKSHOP*XmMenuShell*background	Background color of menus	#DEDEDE
WORKSHOP*XmList*background	Background color of lists, such as the Match list in the Browsing window	#DEDEDE
WORKSHOP*topShadowColor	Color of shadows at top and left edges of buttons and text boxes	#FFFFFF

Scrollbar Background and Toggle Button Colors

TABLE A-17 lists the resources for the colors of the scrollbar background (trough), and the colors in toggle buttons to indicate toggle on or off.

TABLE A-17   Trough and Toggle Buttons Color Resources

Resource Name	Description	Default Value
WORKSHOP*HTML*troughColor	Background color for scrollbars	#DEDEDE
WORKSHOP*XmToggleButton.selectColor	Color for check boxes when selected	#FF9696
WORKSHOP*XmToggleButton.fillOnSelect	Fill check box when selected	true
WORKSHOP*XmToggleButtonGadget.selectColor	Color for radio buttons when selected	#FF9696
WORKSHOP*XmToggleButtonGadget.fillOnSelect	Fill radio button when selected	true

Editable Text Editor Resources

TABLE A-18 through TABLE A-25 describe the text editor resources that you can change in the ESERVE file.

Text Editor Default Path Names

The resources listed in TABLE A-18 are used by the edit server to start the text editor of your choice. If a fully qualified path is specified, it is executed.

TABLE A-18   Text Editor Default Path Resources

Resource Name	Default Value
ESERVE*defaultGnuEmacsPath	emacs
ESERVE*defaultXEmacsPath	xemacs
ESERVE*defaultNEditPath	nedit
ESERVE*defaultGVimPath	gvim

The values for these resources can either be fully qualified paths or the base name of the command (for instance, myfavoriteemacs).

If a base name is used then it is invoked from the PATH environment variable.

Blinking Pointer

TABLE A-19 lists the resource to change the pointer in text editor windows to a non-blinking pointer. Default setting is for a blinking pointer. Set to 0 for a non-blinking pointer.

TABLE A-19   Blinking Pointer Resource

Resource Name	Default Value
ESERVE*DtTerm.blinkRate	250

Fonts for Text Editor Motif Environments

TABLE A-20 lists font resources for the text editor windows that are specific to Motif environments only and are not used by CDE.

TABLE A-20 Motif (non-CDE) Windowing Systems Editor Window Font Resources

Resource Name	Default Value
`ESERVE.labelFontList`	`-*-lucida-medium-r-normal-*-12-*-*-*-*-*-*-*`
`ESERVE.buttonFontList`	`-*-lucida-medium-r-normal-*-12-*-*-*-*-*-*-*`
`ESERVE.textFontList`	`-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-*-*`
`ESERVE*dtTerm*userFont`	`-*-lucidatypewriter-medium-r-normal-*-12-*-*-*-*-*-*-*`

Text Editor Window Colors

TABLE A-21 lists the resource for foreground and background colors in the text editor windows.

TABLE A-21 Editor Windows, Dialog Boxes, Menus, and Buttons Color Resources

Resource Name	Description	Default Value
`ESERVE*foreground`	Foreground color of windows (text such as labels)	`black`
`ESERVE*background`	Background color of windows	`#dedededede`
`ESERVE*XmPushButton*background`	Background color of buttons	`#dedededede`
`ESERVE*XmMenuShell*background`	Background color of menus	`#dedededede`

Scrolling List Background Color

TABLE A-22 lists the resource for the background color for scrolling lists available from a text editor.

TABLE A-22 Scrolling List Background Color Resource

Resource Name	Description	Default Value
`ESERVE*XmList*background`	Background color of scrolling lists	`#dedededede`

Writable Text Area Background Color

TABLE A-23 lists colors for areas in the text editor windows containing text, other than menus and buttons (not applicable to Emacs and XEmacs).

TABLE A-23   Writable Text Area Background Color Resources

Resource Name	Default Value
ESERVE*XmTextField*background	white
ESERVE*XmText*background	white
ESERVE*dtTerm*background	white
ESERVE*readwriteBackground	white

Balloon Expression Evaluator Popup Dimensions

The resource listed in TABLE A-24 sets the maximum dimensions for the balloon expression evaluator popup that instantly shows you the current value of the expression at which your cursor is pointing in your editor. Width is measured in characters, and height is measured in lines.

TABLE A-24   Balloon Expression Evaluator Popup Dimensions Resources

Resource Name	Default Value
ESERVE.balloonWidth	60
ESERVE.balloonHeight	20

Text Editor Audible Warnings

The resource listed in TABLE A-25 enables you to turn off audible warning beeps in the text editor windows. The possible values are -XmBell and -XmNONE.

TABLE A-25   Text Editor Audible Warning Resource

Resource Name	Description	Default Value
ESERVE*audibleWarning	Turns audible beeps on and off	XmBell

The `make` Utility and Makefiles

You can use the `make` utility and makefiles to help automate building of an application with Sun WorkShop. This appendix provides some basic information about the `make` utility, makefiles, and makefile macros. It also refers you to dialog boxes in Sun WorkShop that allow you to set makefile options and to add, delete, and override makefile macros. To build your programs without writing your own makefile, see "Building a Program" on page 26 and "Building With Default Values" on page 27.

The `make` utility applies intelligence to the task of program compilation and linking. Typically, a large application might exist as a set of source files and `INCLUDE` files, which require linking with a number of libraries. Modifying any one or more of the source files requires recompilation of that part of the program and relinking. You can automate this process by specifying the interdependencies between files that make up the application along with the commands needed to recompile and relink each piece. With these specifications in a file of directives, `make` insures that only the files that need recompiling are recompiled and that relinking uses the options and libraries you want.

For more information, there are commercially published books on how to use `make` as a program development tool, including *Managing Projects with `make`*, by Oram and Talbott, from O'Reilly & Associates.

The Makefile

A file called `makefile` tells the `make` utility in a structured manner which source and object files depend on other files. It also defines the commands required to compile and link the files.

Each file to build, or step to perform, is called a *target*. Each entry in a makefile is a rule expressing a target object's dependencies and the commands needed to build or make that object. The structure of a rule in the makefile is:

target: *dependencies-list*
TAB   *build-commands*

For the dependencies, each entry starts with a line that names the target file, followed by all the files the target depends on. For the build commands, each entry has one or more subsequent lines that specify the Bourne shell commands that will build the target file for this entry. Each of these command lines must be indented by a tab character.

Fortran 77 Example

You have a program consisting of the following source files and a makefile:

- makefile
- commonblock
- computepts.f
- pattern.f
- startupcore.f

Both pattern.f and computepts.f have an INCLUDE of commonblock, and you wish to compile each .f file and link the three relocatable files, along with a series of libraries, into a program called pattern.

The makefile contains the following lines.

CODE EXAMPLE B-1   Fortran 77 Makefile

```
pattern: pattern.o computepts.o startupcore.o
    f77 pattern.o computepts.o startupcore.o –lcore77 \
    –lcore –lsunwindow –lpixrect –o pattern
pattern.o: pattern.f commonblock
    f77 –c –u pattern.f
computepts.o: computepts.f commonblock
    f77 –c –u computepts.f
startupcore.o: startupcore.f
    f77 –c –u startupcore.f
```

The first line of this makefile indicates that making `pattern` depends on `pattern.o`, `computepts.o`, and `startupcore.o`. The next line and its continuations give the command for making `pattern` from the relocatable `.o` files and libraries.

C++ Example

You have a program consisting of the following source files and a makefile:

- `manythreads.cc`
- `Makefilemany.cc`
- `thr.cc`
- `misc.h`
- `defines.h`

The target files are:

- `many`
- `manythreads`
- `thrI`

The makefile contains the following lines.

CODE EXAMPLE B-2   C++ Makefile

```
all: many manythreads thrI
many: many.cc
    CC -o many many.cc -g -D_REENTRANT -lm -lnsl -lsocket -lthread
thrI: thr.cc
    CC -o thrI thr.cc -g -D_REENTRANT -lm -lnsl -lsocket -lthread
manythreads: manythreads.cc
    CC -o manythreads -g -D_REENTRANT manythreads.cc -lnsl \
        -lsocket -lthread
```

The first line of this makefile groups a set of targets with the label `all`. The succeeding lines give the commands for making the three targets, each of which has a dependency on one of the source files.

The make Utility

To start the make utility, type the following at a command line:

```
%  make
```

You can add a number of options to the make command for your application using the Build Options dialog box in Sun WorkShop (see "Specifying Build Options" on page 27).

The make utility looks for a file named makefile or Makefile in the current directory and takes its instructions from that file.

The make utility:

1. Reads makefile to determine all the target files it must process, the files they depend on, and the commands needed to build them

2. Finds the date and time each file was last changed

3. Rebuilds any target file that is older than any of the files it depends on, using the commands from makefile for that target

To make writing a makefile easier, the make utility has default rules that it uses depending on the suffix of a target file. Recognizing the .f suffix, make uses the f77 compiler, passing as arguments any flags specified by the FFLAGS macro, the -c flag, and the name of the source file to be compiled.

CODE EXAMPLE B-3 demonstrates this rule twice.

CODE EXAMPLE B-3   make Default Suffix Rule

```
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
    f77 $(OBJ) -lcore77 -lcore -lsunwindow \
   -lpixrect -o pattern
pattern.o: pattern.f commonblock
    f77 $(FFLAGS) -c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

make uses default rules to compile computepts.f and startupcore.f. Similarly, the suffix rules for .f95 files invoke the f95 compiler.

Macros

The `make` utility's macro facility allows simple parameterless string substitutions. For example, the list of relocatable files that make up the target program `pattern` can be expressed as a single macro string, making it easier to change. See also the `make`(1S) man page for information about `make` macros.

A macro string definition has the form:

```
% make NAME=string
```

Use of a macro string is indicated by $(*NAME*), which is replaced by `make` with the actual value of the macro string named.

This example adds a macro definition naming all the object files to the beginning of `makefile`:

```
OBJ=pattern.o computepts.o startupcore.o
```

Now the macro can be used in both the list of dependencies as well as on the `f77` link command for target `pattern` in `makefile`:

```
pattern: $(OBJ)
    f77 $(OBJ) -lcore77 -lcore -lsunwindow \
    -lpixrect -o pattern
```

For macro strings with single-letter names, the parentheses can be omitted.

You can use the Make Macros dialog box in Sun WorkShop to add macros to or delete macros from the Macros list in your WorkShop target and reassign values for makefile macros in the list. For more information, see "Using Makefile Macros" on page 28.

The initial values of makefile macros can be overridden with command-line options to `make`. For example, you have the following line at the top of `makefile`:

```
FFLAGS=-u
```

You also have the compile-line of `computepts.f`:

```
f77 $(FFLAGS) –c computepts.f
```

You have the final link:

```
f77 $(FFLAGS) $(OBJ) –lcore77 –lcore –lsunwindow \
    lpixrect –o pattern
```

A `make` command without arguments uses the value of `FFLAGS` set above. However, this can be overridden from the command line:

```
% make "FFLAGS=–u –O"
```

The definition of the `FFLAGS` macro on the `make` command line overrides the `makefile` initialization, and both the `–O` flag and the `–u` flag are passed to `f77`. `FFLAGS=` can also be used on the command line to reset the macro so that it has no effect.

The `dmake` Utility

This appendix describes the way the distributed make (`dmake`) utility distributes builds over several hosts to build programs concurrently over a number of workstations or multiple CPUs. See also the dmake(1) man page.

Basic Concepts

Distributed make (`dmake`) is a superset of the `make` utility and allows you to concurrently distribute the process of building large projects, consisting of many programs, over a number of workstations and, in the case of multiprocessor systems, over multiple CPUs.

You execute `dmake` on a `dmake` host and distribute jobs to build servers. You can also distribute jobs to the `dmake` host, in which case it is also considered to be a build server. The `dmake` utility distributes jobs based on makefile targets that `dmake` determines (based on your makefiles) can be built concurrently. From the `dmake` host you can control which build servers are used and how many `dmake` jobs are allotted to each build server. The number of `dmake` jobs that can run on a given build server can also be limited on that server.

The distribution of `dmake` jobs is controlled in two ways:

1. A `dmake` user on a `dmake` host can specify the machines to use as build servers and the number of jobs to distribute to each build server.

2. The owner of a build server (a user who can alter the `/etc/opt/SPROdmake/dmake.conf` build server configuration file) can control the maximum total number of `dmake` jobs that can be distributed to that build server.

> **Note –** If you access dmake from the Building window, see the online help for information about specifying your build servers and jobs. If you access dmake from the command line, see the dmake(1) man page.

To understand dmake, you should know about:

- The dmake host
- The build server

The dmake Host

The dmake host is defined as the machine on which the dmake command is initially issued. The dmake utility searches for a runtime configuration file to determine where to distribute jobs. Generally, this file must be in your home directory on the dmake host and is named .dmakerc. The dmake utility searches for the runtime configuration file in these locations and in the following order:

1. The path name you specify on the command line using the -c option

2. The path name you specify using the DMAKE_RCFILE makefile macro

3. The path name you specify using the DMAKE_RCFILE environment variable

4. $(HOME)/.dmakerc

If a runtime configuration file is not found, the dmake utility distributes two jobs to the dmake host.

The runtime configuration file allows you to specify a list of build servers and the number of jobs you want distributed to each build server. CODE EXAMPLE C-1 is an example of a .dmakerc file.

CODE EXAMPLE C-1   .dmakerc File

```
# My machine. This entry causes dmake to distribute to it.
falcon  { jobs = 1 }
hawk
eagle   { jobs = 3 }
# Manager's machine. She's usually at meetings
heron   { jobs = 4 }
avocet
```

The entries falcon, hawk, eagle, heron, and avocet are listed build servers. You can specify the number of jobs you want distributed to each build server. The default number of jobs is two. Any line that begins with the # character is interpreted as a

comment. In the example above, the list of build servers includes `falcon` which is also the `dmake` host. The `dmake` host can also be specified as a build server. If you do not include it in the runtime configuration file, no `dmake` jobs are distributed to it.

You can also construct groups of build servers in the runtime configuration file. The `dmake` utility provides you with the flexibility of easily switching between different groups of build servers as circumstances warrant. For instance, you may define groups of build servers for builds under different operating systems, or you may define groups of build servers that have special software installed on them.

CODE EXAMPLE C-2 shows a `.dmakerc` file that contains groups of build servers.

CODE EXAMPLE C-2   `.dmakerc` File With Groups of Build Servers

```
earth   { jobs = 2 }
mars    { jobs = 3 }

group lab1 {
        host falcon{ jobs = 3 }
        host hawk
        host eagle { jobs = 3 }
}

group lab2 {
        host heron
        host avocet{ jobs = 3 }
        host stilt { jobs = 2 }
}

group labs {
        group lab1
        group lab2
}

group sunos5.x {
        group labs
        host jupiter
        host venus{ jobs = 2 }
        host pluto { jobs = 3 }
}
```

Formal groups are specified by the `group` keyword and lists of their members are delimited by braces ({}). Build servers that are members of groups are specified by the optional `host` keyword. Groups can be members of other groups. Individual build servers can be listed in runtime configuration files that also contain groups of build servers; in this case, `dmake` treats these build servers as members of the unnamed group.

In order of precedence, the dmake utility distributes jobs to the following:

1. The formal group specified on the command-line as an argument to the --g option

2. The formal group specified by the DMAKE_GROUP makefile macro

3. The formal group specified by the DMAKE_GROUP environment variable

4. The first group specified in the runtime configuration file

The dmake utility allows you to specify a different execution path for each build server. By default dmake looks for the dmake support binaries on the build server in the same logical path as on the dmake host. You can specify alternate paths for build servers as a host attribute in the .dmakerc file. For example:

CODE EXAMPLE C-3   .dmakerc File With Alternate Paths for Build Servers

```
group lab1 {
        host falcon{ jobs = 10 , path = "/set/dist/sparc-S2/bin" }
        host hawk{ path = "/opt/SUNWspro/bin"                    }
}
```

You can use double quotation marks to enclose the names of groups and hosts in the .dmakerc file. This allows you more flexibility in the characters that you can use in group names. Digits are allowed, as well as alphabetic characters. Names that start with digits should be enclosed in double quotes. For example:

CODE EXAMPLE C-4   .dmakerc File With Special Characters

```
group "123_lab" {
        host "456_hawk"{ path = "/opt/SUNWspro/bin"             }
}
```

The Build Server

Each build server that is to participate in a distributed build must have a file called /etc/opt/SPROdmake/dmake.conf. This file is the build server configuration file and specifies the maximum total number of dmake jobs that can be distributed to that particular build server by all dmake users. In addition, it might specify the nice priority under which all dmake jobs should run.

Note – If the /etc/opt/SPROdmake/dmake.conf file does not exist on a build server, no dmake jobs will be allowed to run on that server.

CODE EXAMPLE C-5 is an example of an `/etc/opt/SPROdmake/dmake.conf` file. This file sets the maximum number of dmake jobs permitted to run on a build server (from all dmake users) to be eight (8).

CODE EXAMPLE C-5   `dmake.conf` File

```
max_jobs: 8
nice_prio: 5
```

You can use a machine as a build server if it meets the following requirements:

- From the dmake host (the machine you are using), you must be able to use rsh without being prompted for a password to remotely execute commands on the build server. See the rsh(1) man page. For example:

```
% rsh build-server which dmake
/opt/SUNWspro/bin/dmake
```

- The bin directory in which the dmake software is installed must be accessible from the build server. It is common practice to have all build servers share a common dmake installation directory. See the share(1M) and mount(1M) man pages.

- By default, dmake assumes that the logical path to the dmake executables on the build server is the same as on the dmake host. You can override this assumption by specifying a path name as an attribute of the host entry in the runtime configuration file. For example:

```
group sparc-cluster {
    host wren   { jobs = 10 , path = "/export/SUNWspro/bin"}
    host stimpy { path = "/opt/SUNWspro/bin"                }
```

- The source hierarchy you are building must be accessible from the build server and mounted under the same name.

Impact of the `dmake` Utility on Makefiles

To run a distributed make, use the executable file `dmake` in place of the standard `make` utility. You should understand the Solaris `make` utility before you use `dmake`. If you need to read more about the `make` utility, see the Solaris Programming Utilities Guide and the `make`(1) man page. If you use the `make` utility, the transition to `dmake` requires little or no alteration.

The methods and examples shown in this section present the kinds of problems that lend themselves to being solved with `dmake`. This section does not suggest that any one approach or example is the best.

As procedures become more complicated, so do the makefiles that implement them. The examples in this section illustrate common code-development predicaments and some straightforward methods to simplify them using `dmake`.

If you use a makefile template from the outset of your project, custom makefiles that evolve from the makefile templates will be more familiar, easier to understand, easier to integrate, easier to maintain, and easier to reuse.

Concurrent Building of Targets

Large software projects typically consist of multiple independent modules that can be built concurrently. The `dmake` utility supports concurrent processing of targets on multiple machines over a network. This concurrency can markedly reduce the time required to build a large project.

When given a target to build, `dmake` checks the dependencies associated with that target, and builds those that are out of date. Building those dependencies may, in turn, entail building some of their dependencies. When distributing jobs, `dmake` starts every target that it can. As these targets complete, `dmake` starts other targets. Nested invocations of `dmake` are not run concurrently by default, but this can be changed (see "Parallelism" on page 96 for more information).

Since `dmake` builds multiple targets concurrently, the output of each build is produced simultaneously. To avoid intermixing the output of various commands, `dmake` collects output from each build separately. The `dmake` utility displays the commands before they are executed. If an executed command generates any output, warnings, or errors, `dmake` displays the entire output for that command. Since commands started later might finish earlier, this output might be displayed in an unexpected order.

Limitations on Makefiles

Concurrent building of multiple targets places some restrictions on makefiles. Makefiles that depend on the implicit ordering of dependencies might fail when built concurrently. Targets in makefiles that modify the same files may fail if those files are modified concurrently by two different targets. Some examples of possible problems are discussed in this section.

Dependency Lists

When building targets concurrently, it is important that dependency lists be accurate. For example, if two executables use the same object file but only one specifies the dependency, then the build may cause errors when done concurrently. For example, consider the following makefile fragment:

```
all: prog1 prog2
prog1: prog1.o aux.o
    $(LINK.c) prog1.o aux.o -o prog1
prog2: prog2.o
    $(LINK.c) prog2.o aux.o -o prog2
```

When built serially, the target `aux.o` is built as a dependent of `prog1` and is up-to-date for the build of `prog2`. If built in parallel, the link of `prog2` can begin before `aux.o` is built and is therefore incorrect. The `.KEEP_STATE` feature of `make` detects some dependencies, but not the one shown above.

Explicit Ordering of Dependency Lists

Other examples of implicit ordering dependencies are more difficult to fix. For example, if all of the headers for a system must be constructed before anything else is built, then everything must be dependent on this construction. This causes the makefile to be more complex and increases the potential for error when new targets are added to the makefile. The user can specify the special target `.WAIT` in a makefile to indicate this implicit ordering of dependents. When `dmake` encounters the `.WAIT` target in a dependency list, it finishes processing all prior dependents before proceeding with the following dependents. More than one `.WAIT` target can be used in a dependency list. The following example shows how to use `.WAIT` to indicate that the headers must be constructed before anything else.

```
all: hdrs .WAIT libs functions
```

You can add an empty rule for the `.WAIT` target to the makefile so that the makefile is compatible with the `make` utility.

Concurrent File Modification

You must make sure that targets built concurrently do not attempt to modify the same files at the same time. This can happen in a variety of ways. If a new suffix rule is defined that must use a temporary file, the temporary file name must be different for each target. You can accomplish this by using the dynamic macros `$@` or `$*`. For example, a `.c.o` rule that performs some modification of the `.c` file before compiling it might be defined as:

```
.c.o:
    awk -f modify.awk $*.c > $*.mod.c
    $(COMPILE.c) $*.mod.c -o $*.o
    $(RM) $*.mod.c
```

Concurrent Library Update

Another potential concurrency problem is the default rule for creating libraries that also modifies a fixed file, that is, the library. The inappropriate `.c.a` rule causes `dmake` to build each object file and then archive that object file. When `dmake` archives two object files in parallel, the concurrent updates will corrupt the archive file.

```
.c.a:
    $(COMPILE.c) -o $% $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

A better method is to build each object file and then archive all the object files after completion of the builds. An appropriate suffix rule and the corresponding library rule are:

```
.c.a:
    $(COMPILE.c) -o $% $<

lib.a: lib.a($(OBJECTS))
    $(AR) $(ARFLAGS) $(OBJECTS)
    $(RM) $(OBJECTS)
```

Multiple Targets

Another form of concurrent file update occurs when the same rule is defined for multiple targets. An example is a yacc(1) program that builds both a program and a header for use with lex(1). When a rule builds several target files, it is important to specify them as a group using the + notation. This is especially so in the case of a parallel build.

```
y.tab.c y.tab.h: parser.y
    $(YACC.y) parser.y
```

This rule is actually equivalent to the two rules:

```
y.tab.c: parser.y
    $(YACC.y) parser.y
y.tab.h: parser.y
    $(YACC.y) parser.y
```

The serial version of make builds the first rule to produce y.tab.c and then determines that y.tab.h is up-to-date and need not be built. When building in parallel, dmake checks y.tab.h before yacc has finished building y.tab.c and notices that y.tab.h *does* need to be built, it then starts another yacc in parallel with the first one. Since both yacc invocations are writing to the same files (y.tab.c and y.tab.h), these files are apt to be corrupted and incorrect. The correct rule uses the + construct to indicate that both targets are built simultaneously by the same rule. For example:

```
y.tab.c + y.tab.h: parser.y
    $(YACC.y) parser.y
```

Parallelism

Sometimes file collisions cannot be avoided in a makefile. An example is `xstr(1)`, which extracts strings from a C program to implement shared strings. The `xstr` command writes the modified C program to the fixed file `x.c` and appends the strings to the fixed file `strings`. Since `xstr` must be run over each C file, the following new `.c.o` rule is commonly defined:

```
.c.o:
    $(CC) $(CPPFLAGS) -E $*.c | xstr -c -
    $(CC) $(CFLAGS) $(TARGET_ARCH) -c x.c
    mv x.o $*.o
```

The `dmake` utility cannot concurrently build targets using this rule since the build of each target writes to the same `x.c` and `strings` files. Nor is it possible to change the files used. You can use the special target `.NO_PARALLEL:` to tell `dmake` not to build these targets concurrently. For example, if the objects being built using the `.c.o` rule were defined by the `OBJECTS` macro, the following entry would force `dmake` to build those targets serially:

```
.NO_PARALLEL: $(OBJECTS)
```

If most of the objects must be built serially, it is easier and safer to force all objects to default to serial processing by including the `.NO_PARALLEL:` target without any dependents. Any targets that can be built in parallel can be listed as dependencies of the `.PARALLEL:` target:

```
.NO_PARALLEL:
.PARALLEL: $(LIB_OBJECT)
```

When `dmake` encounters a target that invokes another `dmake` command, it builds that target serially, rather than concurrently. This prevents problems where two different `dmake` invocations attempt to build the same targets in the same directory. Such a problem might occur when two different programs are built concurrently, and each must access the same library. The only way for each `dmake` invocation to be sure that the library is up-to-date is for each to invoke `dmake` recursively to build that library. The `dmake` utility recognizes a nested invocation only when the `$(MAKE)` macro is used in the command line.

If you nest commands that you know will not collide, you can force them to be done in parallel by using the `.PARALLEL:` construct.

When a makefile contains many nested commands that run concurrently, the load-balancing algorithm may force too many builds to be assigned to the local machine. This may cause high loads and possibly other problems, such as running out of swap space. If such problems occur, allow the nested commands to run serially.

Source Browsing With `sbquery`, `sb_init`, and `sbtags`

This appendix:

- Describes `sbquery`, one of the command-line utilities for browsing source code
- Tells you how to work with source files where database information is stored in multiple directories
- Describes the `sbtags` command, which provides a quick and convenient method for collecting browsing information from source files

The information in this chapter pertains mainly to the use of the command line to complete tasks also available from within Sun WorkShop. For more conceptual information on using source browsing, see Chapter 5 and the Browsing Source Code section of the online help (you can access the online help through the Help menu in any Sun WorkShop window).

The `sbquery` Utility

The `sbquery` utility provides you with a command-line browsing environment that you can access from terminals and from workstations emulating terminals. By default, `sbquery` searches for symbols in the database in the current working directory. If you want to browse a database stored in another directory, see "The `sb_init` File and Commands" on page 103.

To issue a query from the command line, type `sbquery`, followed by any command-line options and their arguments, followed by the symbol you want to search for:

% **sbquery** [*options*] *symbols*

sbquery displays a list of matches that includes the file in which the symbol appears, the line number, the function containing the symbol, and the source line containing the symbol.

Options

To obtain a list of the sbquery command-line options, type sbquery at the shell prompt. TABLE D-1 lists and describes the options (see also the sbquery(1) man page).

TABLE D-1   sbquery Options

Arguments	Description
-pattern *symbol*	Queries on *symbol*, which may contain special characters, including a leading dash (-). This option allows you to query on a symbol that looks like a command-line option. For instance, you can query on the symbol -help, and sbquery distinguishes it from the regular option -help.
-break_lock	Breaks the lock on a locked database. This argument might be needed if the update of the index file is aborted. The next time you issue a query you might get a message telling you that the database is locked. After using this option, your database may be in an inconsistent state. To ensure consistency, remove the database directory and recompile your program.
-files_only	Lists only the files where the symbols you are searching for appear.
-help	Displays a synopsis of the sbquery command. Equivalent to typing sbquery with no options and no symbol.
-help_focus	Displays a list of the focus options available for querying only specific program types in a directory. Use focus options (see TABLE D-3) to issue a query limited to specific units of code such as programs or functions.
-help_filter *language*	Displays a list of the languages for which filter options are available for -help_filter. Displays a list of the filter options for the language for -help_filter *language*. Use filter options (see TABLE D-2) to search for symbols based on how they are used in a program.

Arguments	Description
max_memory *size*	Sets the approximate amount of memory in megabytes that should be allocated before sbquery uses temporary files when building the index file.
-no_case	Makes the query case-insensitive.
-no_source	Displays only the file name and line number associated with each match and not the source line containing the match.
-no_update	Does not rebuild the index file when you issue a query following compilation. If you do not include this option and issue a query following compilation or recompilation, then the database updates and processes your query.
-o *file*	Sends query output to *file*.
-show_db_dirs	Lists all database directories scanned when you issue a query. The list includes the following: the database directory in the current working directory and all other database directories specified by the import or export commands in your sb_init file.
-symbols_only	Displays a list of all symbols that match the patterns in your search pattern. This is useful when you use wildcards in a query.
-version	Displays the current version number.
-sh_pattern	Uses shell-style expressions when issuing a query that includes wildcards. This wildcard setting is the default; include this option if you are doing other pattern matching on the same command line. See the sh(1) man page for more information about shell-style pattern matching.
-reg_expr	Uses regular expressions when issuing a query that includes wildcards. If you do not include this option, shell-style patterns are assumed.
-literal	Uses only literal strings and does not use any wildcard expressions for the query. This is useful when you want to search for a string that contains meta characters from other wildcard schemes.

Two types of options are available to help you narrow your search: filter options described in TABLE D-2 and focus options described in TABLE D-3.

The filter options are used to search for symbols based on how they are used in a program. For example, you could limit your search to declarations of variables.

```
% sbquery -help_filter language
```

TABLE D-2   Filter Language Options

Filter Option	Description
ansi_c	C
sun_as	Assembly language
sun_c_plus_plus	C++
sun_f77	Fortran 77
sun_f90	Fortran 90

The focus options listed in TABLE D-3 limit your search to specific classes of code, such as particular programs, functions, or libraries.

```
% sbquery focus-option symbol
```

TABLE D-3   Focus Options

Focus Option	Description
-in_program *program*	Limits query to matches in *program*.
-in_directory *directory*	Limits query to matches in *directory*.
-in_source_file *source-file*	Limits query to matches in *source-file*.
-in_function *function*	Limits query to matches in *function*.
-in_class *class*	Limits query to matches in *class*.
-in_template *template*	Limits query to matches in *template*.
-in_language *language*	Limits query to matches in *language*.

Note – If you include two or more focus options, a match is returned if it is found with any of the supplied focus options.

Environment Variables

Environment variables provide information that affects the operation of `sbquery` (and source browsing in Sun WorkShop).

TABLE D-4 Environment Variables

Variable	Description
HOME	The name of your login directory.
PWD	The full path name of the current directory.
SUNPRO_SB_ATTEMPTS_MAX	The maximum number of times the index builder tries to access a locked database.
SUNPRO_SB_EX_FILE_NAME	The absolute path name of the `sun_source_browser.ex` file.
SUNPRO_SB_INIT_FILE_NAME	The absolute path name of the `sb_init` file. For more information on `sb_init`, see "The `sb_init` File and Commands" on page 103.

The `sb_init` File and Commands

This section describes how to work with source files where database information is stored in multiple directories. As a default, the database is built in the current working directory and searches that database when you issue a query.

The text file `sb_init` is used by the Source Browsing mode of Sun WorkShop, the compilers, and `sbtags` to obtain control information about the source browsing database structure. Use `sb_init` if you want to work with source files whose database information is stored in multiple directories.

The `sb_init` file should be placed in the `SunWS_config` directory, which should be placed in the directory from which source browsing, the compilers, and `sbtags` are run. These tools look in the current working directory for the `sb_init` file.

The default is to look in the current working directory for the `sb_init` file. To instruct Sun WorkShop and the compiler to search for the `sb_init` file in another directory, set the environment variable `SUNPRO_SB_INIT_FILE_NAME` to *absolute-pathname*/`sb_init`.

To use an sb_init command, add the command to the sb_init file. The sb_init file is limited to the following commands:

TABLE D-5   sb_init Commands

Comand	Description
import	Reads databases in directories other than the current working directory.
export	Writes database component files associated with specified source files to directories other than the current working directory of the compiler. This command also acts as an import command.
replacepath	Specifies how to modify paths to file names found in the database, allowing you to move a database.
automount-prefix	Enables you to browse source files on a machine other than the one on which you compiled your program.
cleanup-delay	Limits the time elapsed between rebuilding the index and the associated database garbage collection.

import

```
% import pathname
```

This command allows the Source Browsing mode of Sun WorkShop to read databases in directories other than the current working directory. Use of the import command enables you to retain separate databases for separate directories.

For example, you may want to set up administrative boundaries so that programmers working on Project A cannot write into directories for Project B and vice versa. In that case, Project A and Project B each need to maintain their own databases, both of which can then be read but not written by programmers working on the other project.

export

```
% export prefix into path
```

This command causes the compilers and sbtags to write database component files associated with source files to directories other than the current working directory used by the Source Browsing mode of Sun WorkShop and the compiler.

Whenever the compiler processes a source file whose absolute path starts with *prefix*, the resulting browser database (`.bd`) file is stored in *path*.

The export command contains an implied `import` command of *path*, so that exported database components are automatically read by the Source Browsing mode of Sun WorkShop.

The `export` command enables you to save disk space by placing database files associated with identical files, such as `#include` files from `/usr/include`, in a single database, while still retaining distinct databases for individual projects.

If your `sb_init` files include multiple `export` commands, then you must arrange them from the most specific to the least specific. The compiler scans `export` commands in the same order that it encounters them in the `sb_init` file.

replacepath

% **replacepath** *from-prefix* *to-prefix*

This command specifies how to modify path names in the source browsing database.

In general, *from-prefix* corresponds to the automounter mount point (the path name where the automounter actually mounts the file system); *to-prefix* corresponds to the automounter trigger point (the path name known and used by the developer).

There is considerable flexibility in how an automounter is used; the method can vary from host to host.

Path replacement rules are matched in the order that they are found, and matching stops after a replacement is done.

The default `replacepath` command is used to strip away automounter artifacts:

% **replacepath** */tmp_mnt*

When used for this purpose, the command should be given with the mount point as the first argument and the trigger point as the second argument.

automount-prefix

```
% automount-prefix mount-point  trigger-point
```

The `automount-prefix` command enables you to browse on a machine other than the one on which you compiled your program. This command is identical to the `replacepath` command except that `automount-prefix` path translations occur at compile time and are written into the database.

The `automount-prefix` command defines which automounter prefixes to remove from the source names stored in the database. The directory under which the automounter mounts the file systems is the *mount-point*; the *trigger-point* is the prefix you use to access the exported file system. The default is `/`.

If the path in the database fails, the path translations from both commands (that is, `automount-prefix` and `replacepath`) are used to search for source files while browsing.

At first glance, searching both paths may not seem possible; the browser database that is created when you run the compiler contains the absolute path for each source file. If the absolute path is not uniform across machines, then Sun WorkShop will not be able to display the source files when it responds to a query.

To get around this problem, you can do one of the following:

- Ensure that all source files are mounted at the same mount point on all machines.
- Compile your programs in an automounted path. A reference to such a path causes the automounter to automatically mount a file system from another machine.

There is a default `automount-prefix` command that is used to strip away automounter artifacts:

```
% automount-prefix /tmp-mnt  /
```

The default rule is generated only if no `automount-prefix` commands are specified.

For more information on using the automounter, see the `automount(1M)` man page.

cleanup-delay

This command limits the time elapsed between rebuilding the index and the associated database garbage collection. The compilers automatically invoke `sbcleanup` if the limit is exceeded. The default value is 12 hours.

The `sbtags` Utility

The `sbtags` command provides a method for collecting browsing information from source files, enabling you to collect minimal browsing information for programs that do not compile completely. See also the `sbtags(1)` man page.

The `sbtags` command collects a subset of the information available through compilation. The reduced information restricts some browsing functionality. A database generated by `sbtags` enables you to perform queries on functions and variables and to display the function call graph.

A tags database:

- Cannot issue queries about local variables
- Cannot browse classes
- Cannot graph class relationships
- Has limited ability to issue complex queries
- Has limited ability to focus queries
- Has less reliability than compiled information

Once a file has been changed, it often need not be scanned again to incorporate changes into the database.

An `sbtags` database is based on a lexical analysis of the source file. Though it does not always correctly identify all the language constructs, it will operate on files that will not compile and is faster than recompilation.

`sbtags` recognizes definitions for types and functions. It also collects information on function calls. No other information is collected (in particular, other semantic information for complex queries is not collected).

The functionality of `sbtags` is similar to `ctags` and `etags`, except for the Call Grapher information. You may mix direct queries to the database for definitions and graphing with pattern-matching queries.

With an `sbtags` generated database:

- Class Browser and Class Graph features are not available.
- The database does not contain information on all symbols and strings. It contains information on functions, classes, types, and calls to functions.
- Time is saved since the `sbtags` program runs faster than the compiler.
- The database size is much smaller than the size of your source code.
- The database content is not guaranteed to be semantically correct because the `sbtags` program performs only simple syntactic and semantic analysis and may sometimes be in error.

- A database is generated even if the source code cannot be compiled because the code is incomplete or semantically incorrect.

To generate a browsing database using sbtags, type the following at a command line:

```
% sbtags file
```

file is the file for which you want to generate the database. See the sbtags(1) man page for more information.

Glossary

build command	The command that starts the `make` utility, which reads the makefile and builds the make targets.
build directory	The directory from which the build process is started and also the default directory for the makefile.
data display	A feature of the debugging service that allows you to watch the changes in the value of an expression during program execution.
data history	A feature of the debugging service that allows you to evaluate expressions and change the value of a variable while debugging a program.
debugging session	A program with an associated debug process. You can debug many programs at the same time using the session manager.
debug mode	A debugging state that allows you to debug your program using the full functionality of the debugging service. See also **quick mode**.
distributed make (`dmake`)	A version of the `make` utility that organizes the build into multiple tasks and distributes those tasks to multiple CPUs and workstations.
makefile	A file that contains entries that tell the `make` utility in a structured manner which source and object files depend on other files. It also defines the commands required to compile and link the files.
make target	An object that the `make` utility knows how to build from the directions (rules) contained in the makefile.
menu picklist	A list of recently used files, targets, programs, projects, or experiments located on Sun WorkShop menus, allowing easy access to your most recently accessed items.
pattern search mode	A mode in the Browsing window that allows you to search source code for any text string, including text embedded within comments. See also **source browsing mode**.
picklist	See **menu picklist**.

project A list of files and compiler, debugger, and build-related options used to build an executable, a static library/archive, a shared library, a Fortran application, a complex application, or a user makefile application.

quick mode A debugging state that allows you to run a program normally but with debugging ready in the background to save the program in case your program terminates abnormally. See also **debug mode**.

run parameters The program arguments, the directory in which the program is run, and any environment variables passed to your program while your program is being debugged.

source browsing mode A mode in the Browsing window that allows you to find all occurrences of any program-defined symbol in your code by searching in a database that is generated when Sun WorkShop compiles your source files with a source browsing option (-xsb). See also **pattern search mode**.

target An object that can be built.

Index

U

unlocking browsing database 100
user makefile
    project 27
    target 24

V

Visual 15

W

`.WAIT` dmake target 93
Welcome to Sun WorkShop dialog box 18
windows
    Breakpoint 36
    Breakpoints 39
    Building 25
    Call Graph 49
    Callers-Callees 58
    Class Browser 52
    Class Graph 50
    Data Display 36
    Dbx Commands 31
    Debugging 31
    WorkShop main 20
WorkShop main window 20