SPARCworks/ProWorks Tutorial

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A
Part No.: 802-3645-10
Revision A, November 1995

Please
Recycle

Adobe PostScript

Contents

Preface

This manual provides exercises that show you how to use the SPARCworks™/ProWorks™ programming environment, which includes the following tools:

- Manager
- MakeTool
- Debugger
- Analyzer (available on SPARCworks/ProWorks running Solaris™ 2.x)
- FileMerge
- SourceBrowser

Before You Begin

This manual is written for application developers who want to use SPARCworks/ProWorks to aid in application development. See the appropriate readme, `sparcworks` or `proworks`, for a description of operating environment requirements in which the SPARCworks/ProWorks toolset runs.

This manual assumes you are familiar with

- Sun® operating system commands and concepts
- The OPEN LOOK® interface and the OpenWindows™ environment, particularly the use of the mouse to activate a window, select text, and click on buttons.

If you are not familiar with the OPEN LOOK interface, see Chapter 1, “Introduction,” or see *Managing the Toolset*.

For more information on the OpenWindows environment, see the *OpenWindows Developer’s Guide: User’s Guide*.

How This Book Is Organized

This manual is organized as follows:

Chapter 1, “Introduction,” describes the sample program Freeway, the structure of the tutorial exercises, and instructions on using the mouse, mouse button operations, and basic window information.

Chapter 2, “Tutorial Exercises,” provides easy to follow exercises that will instruct you on how to use the basic functionality of the tools. You’ll learn how to use the tools by running the sample program, Freeway, that is provided with this tutorial.

- **“Exercise 1 — Starting ProWorks/SPARCworks”** shows you how to start and exit the programming environment.
- **“Exercise 2 — Managing the Toolset”** shows you how to control a working session in SPARCworks/ProWorks, add and delete tools, and how to run non-SPARCworks/ProWorks applications from the Manager.
- **“Exercise 3 — Building a Freeway”** tells you how to build an application using MakeTool and examine makefile statements.
- **“Exercise 4 — Debugging Freeway”** takes you through a debugging process step-by-step using the Debugger. You’ll learn how to set breakpoints, move through the stack, and fix and continue a program.
- **“Exercise 5 — Improving Freeway Performance”** shows you how to use the Analyzer to performance tune an application using the Freeway sample program.

Chapter 3, “The Toolset and the Software Development Cycle,” explains how the toolset contributes to the software development process.

Chapter 4, “The Toolset,” provides a brief overview of each of the tools and their features.

Appendix A, “More About the Debugger,” briefly describes the features available in the Debugger and dbx.

What Typographic Changes and Symbols Mean

The following table describes the typographic conventions and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
♦	A single-step procedure	♦ **Click on the Apply button.**
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	`system%`
$	UNIX Bourne and Korn shell prompt	`system$`
#	Superuser prompt, all shells	`system#`

How to Get Help

ProWorks and SPARCworks products include the following on-line help facilities:

- **AnswerBook®** system displays all the tool manuals. You can read the manual on line and take advantage of dynamically linked headings and cross-references.

  To start the AnswerBook system, type: `answerbook &`
- **Magnify Help™** messages are a standard feature of the OpenWindows software environment. If you have a question, place the pointer on the window, menu, or menu button and press the Help key.
- **Notices** are a standard feature of OPEN LOOK. Some notices inquire about whether or not you want to continue with an action. Others provide information about the end result of an action and appear only when the end result of the action is irreversible.
- **Manual Pages** (man pages) provide information about the command-line utilities of the SunOS operating system. Each tool has at least one man page. To access the man page for FileMerge, type:

  ```
  man filemerge
  ```

  See the man page `dbx(1)` for collecting data commands in batch jobs. See the man page `debugger(1)`for collecting data commands using GUI.

Related Documentation

This manual is part of the tools document set. Other manuals in this set include:

- *Installing SunSoft Developer Products on Solaris*
- *Browsing Source Code*
- *Building Programs with MakeTool*
- *Debugging a Program*
- *Managing the Toolset*
- *Merging Source Files*
- *Performance Tuning an Application*

You can find these and other related documents in the on-line AnswerBook system. To access the AnswerBook system, refer to *Installing SunSoft Developer Products on Solaris*.

Part 1 — Tutorial Exercises

Introduction 1

This tutorial is intended to help you get acquainted with the toolset by walking you through each step of building, debugging, and fine-tuning an application. Before you begin the exercises in Chapter 2, read through this chapter to learn how the tutorial is structured and to learn the mouse and window commands you'll be using throughout the exercises.

This chapter is organized into the following sections:

1.1 Freeway

After you've learned the preliminary tasks of starting the programming environment and running the Manager, you'll use the tutorial's sample program, *Freeway*. Freeway simulates traffic flow on a typical four-lane highway. The vehicles make decisions about how fast to drive based on the conditions you set.

You can create different highway scenarios by setting different conditions per session and noting those settings in the Story window that you can access from the About command in the File menu.

The Freeway sample program is automatically installed along with both SPARCworks and ProWorks tools. You can find the program either in the default directory or in the directory you or your system administrator designates:

- for SPARCworks:

  `/opt/SUNWspro/SW3.1/examples/Freeway`

  /*your_directory*/`SUNWspro/SW3.1/examples/Freeway`

- for ProWorks:

  `/opt/SUNWspro/SW3.1/examples/Freeway`

  /*your_directory*/`SUNWspro/SW3.1/examples/Freeway`

1.2 About the Tutorial

By working through the tutorial, you should learn the basic operations necessary for developing code. Some of the tools, such as the Debugger, have so many features that it would be impossible within the contexts of a tutorial to expose you to all of them. A future section on advanced exercises will cover some of these features, but for detailed information about each of the tools, refer to the individual tool manuals. For an overview of the tools, see Chapter 4, "The Toolset."

It is not the intention of this tutorial to

- Teach you how to program in C or C++
- Provide you with complete information on all the tools

This is a modular tutorial; each exercise consists of a series of tasks that show you how to use the basic features of an individual tool. You can perform each exercise in sequence or go directly to the ones of interest. In certain instances, you are referred to a task in a previous exercise to accomplish a task in another exercise.

Some of the exercises are fairly long, and you will be told which tasks you can stop and continue at a later time. Each exercise begins with a list of goals that covers the fundamental working knowledge you should have of the tool after completing the exercise.

If there are several ways to perform a step, the primary GUI-based method is given first. Alternative methods are provided after the initial task statement. Many operations can be performed in a shell rather than through the graphical user interface. However, this tutorial emphasizes the use of the GUI. Refer to the tool manuals for shell-equivalent commands and instructions.

1.3 Before You Begin

This section gives a quick review of mouse and window operations that you should be familiar with before you begin the tutorial. For more information on mouse and window operations, see *Managing the Toolset*.

1.3.1 Mouse Buttons

You can use a one, two, or three button mouse in the OPEN LOOK user interface. The three standard functions assigned to the buttons are

- SELECT — specifies objects on which to operate and manipulates objects and controls
- ADJUST — extends or reduces a selection
- MENU — displays a menu associated with the pointer location or with a selected object

On a three-button mouse, the left button is SELECT, the middle button is ADJUST, and the right button is MENU.

On a two-button mouse the left button is SELECT and the right button is MENU. ADJUST is carried out with a keyboard equivalent.

On a one-button mouse the button is SELECT. ADJUST and MENU are carried out with keyboard equivalents.

SELECT is the default mouse button; therefore, when you are instructed in the exercises to click or double-click on an object or menu item, click the SELECT button unless explicitly told otherwise.

Figure 1-1 illustrates the mouse button functions.

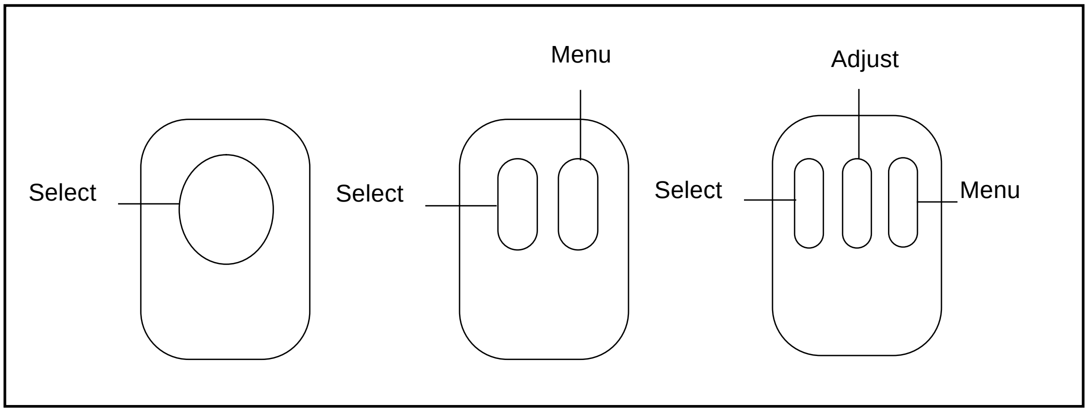


Figure 1-1 Mouse Button Functions

1.3.2 Mouse Button Operations

This tutorial uses the following terms to describe mouse button and pointer operations with the OPEN LOOK user interface:

- **Press —** Push a mouse button and hold it.
- **Click —** Push and release a mouse button.
- **Double-click —** Push and release a mouse button twice in quick succession.
- **Drag —** Push a mouse button and hold it down while moving the pointer.

1.3.3 Pinned and Unpinned Menus

Menus in the OPEN LOOK GUI can either be *pinned* or *unpinned*. An unpinned menu is dismissed as soon as an operation is carried out; a pinned menu remains on the screen until you unpin it.

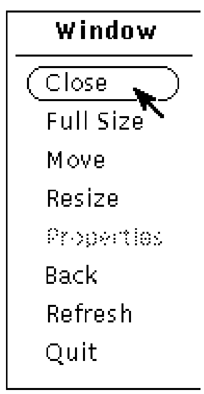
To pin or unpin a menu, place the pointer on the pin and click. With a pinned menu, the pushpin is pushed in. In an unpinned menu, the pushpin is pulled out.

1.3.4 Moving Windows and Menus

You can move and reposition windows and menus. To move and reposition a window or menu, place the pointer in the header or on a corner of the window or menu, press SELECT, and drag the window to the new position. When the window or menu is in position, release SELECT.

1.3.5 Window Menu

The Window menu items activate window operations, such as closing the window, enlarging it, displaying a Properties sheet, moving an application into the background, refreshing the screen, or quitting the application.



To display the Window menu, place the pointer anywhere in the Header and press MENU.

Since Close is the default option on the Window menu, you can quickly close the base window to an icon without displaying the menu. Simply place the cursor on the Window menu button and click.

To reopen the base window, place the pointer on the icon and double-click.

1.4 Resizing and Scaling a Window

You can change the size of an OPEN LOOK base window or pop-up window by resizing and scaling.

1.4.1 Resizing a Window

Resizing a window changes its dimensions without retaining the proportions of the window. To resize a window, place the pointer on a resize corner, press SELECT, and drag the resize corner until the window is the size you want.

1.4.2 Scaling a Window

Scaling changes the size of a window while maintaining its proportions. To scale a window, choose Properties from the Window menu and use the Base Window Scale option. In applications without a Properties window, scaling may not be provided as a feature.

1.5 Getting Help in the OPEN LOOK GUI

The OPEN LOOK GUI provides Magnify Help, a standard feature of the OpenWindows software environment. To access Magnify Help, place the pointer on the window, menu, or menu button, and then press the Help key.

Tutorial Exercises 2

This chapter is organized into the following sections:

The exercises in this chapter cover the following tools:

- Manager—Access all the tools from a tool palette and control your programming sessions.
- MakeTool—Monitor application builds and examine makefiles.
- Debugger—Access an extensive array of debugging features including runtime checking, event management, exception handling and more in addition to the standard debugging operations, such as setting breakpoints and examining the call stack.
- Analyzer—Examine performance data collected from a running application so that you can pinpoint areas of the application needing improvement.

Exercises for the Source Browser are not included in this tutorial. Instructions on using the Source Browser are given in the Source Pane of the Source Browser's main window. Type `sbrowser &` at a shell prompt to start the tool.

2.1 Exercise 1 — Starting ProWorks/SPARCworks

You begin the tutorial by starting the programming environment, which brings up the Manager tool. Once you have learned how to start the ProWorks or SPARCworks toolset, you'll have quick access to the complete toolset. Instead of typing specific tool names at a shell prompt, you can start the tools you want directly from the Manager palette.

Goals — From this exercise you learn how to

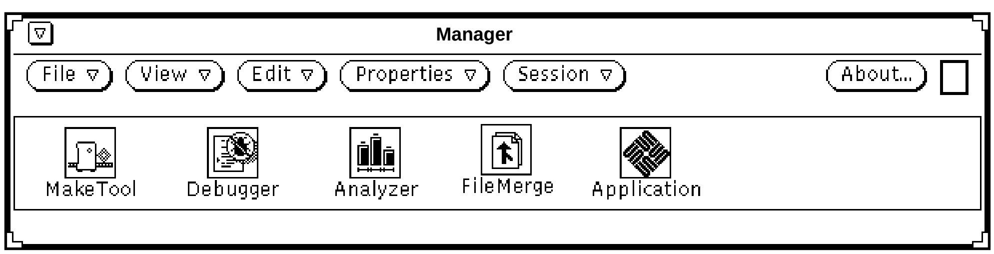
- Start the programming environment (the Manager)
- Open and close the Manager palette
- Check the version number
- Quit the programming environment

1. **Start ProWorks or SPARCworks by typing the following command at the shell prompt.**

   For ProWorks: `proworks &`

   For SPARCworks: `sparcworks &`

   If you've installed ProWorks/SPARCworks correctly, the Manager palette should appear on your screen. If the palette does not appear, check to see if the installation was done correctly and that you have included the pathname in your PATH. The ProWorks and SPARCworks Manager palettes are identical except for the product name in the header.

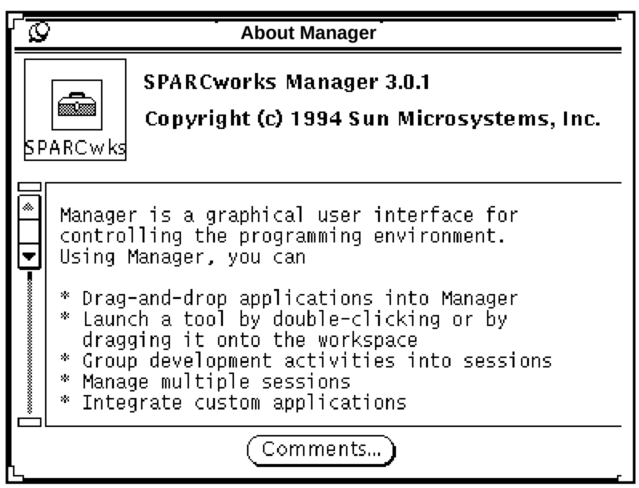


Note – When you bring up the Manager palette, you see the full palette display; for the purposes of this tutorial, the palette figures will show only a single row of tools.

The Freeway sample program that you'll be using later on in the tutorial was automatically installed. However, because it isn't a ProWorks/SPARCworks application, you won't see an icon for it in the Manager palette. The palette does contain a generic application icon that you can assign to an application of your choice (see Section 2.2.3, "Adding and Deleting Tools," on page 2-17).

2. **Close the palette to an icon by clicking on the window menu button. (Refer to Chapter 1 for mouse and window operations.)**
   To reopen the palette, double-click on the icon.

3. **Check that you have the latest version of the toolset by clicking on the About button.**
   A popup window should appear displaying the Manager icon, the version number of the tool, the copyright date, and a brief description of the tool:



   You should have the 3.0.1 Manager running; if you have an earlier version, see your system administrator. You can close the About box by clicking on the pushpin.

4. **Quit Manager by choosing Quit from your window menu.**
   See Section 1.3.5, "Window Menu," on page 1-7 for information on the window menu.

You are done with Exercise 1. You now know how to start and quit Manager, find which version of the toolset you are running, and how to open and close the palette.

The next exercise shows you how to add a non-ProWorks or non-SPARCworks tool to the palette and how to manage the toolset.

2.2 Exercise 2 — Managing the Toolset

The Manager enables you to open or close all the tools you run during a single programming session singly or simultaneously. If you're working on several projects at once, you can open one Manager for each project. It also lets you free up workspace by hiding the tools you have open. Thus, you could work on one project while another is running but hidden from view, freeing up space on the screen and avoiding the confusion of determining which tools to work in.

Goals — From this exercise you learn how to

- Open and close a set of tools
- Change the path of the Manager
- Drag and drop a non-ProWorks/SPARCworks application
- Add a tool to the palette
- Delete a tool from the palette
- Customize the Manager

2.2.1 Controlling a Programming Session

In this task, you control a programming session by starting several tools from the Manager and then closing, opening, and hiding all the tools jointly.

1. **Start a programming session.**

   a. **Open the Manager palette.**
   Either double-click on the toolset icon or restart the Manager if you quit it in the last exercise.

   b. **Start the Analyzer, MakeTool, and Debugger by double-clicking on their icons.**
   After the tools have been started, bring the Manager window to the foreground by clicking in the window header bar.

   Another way to start a tool is by dragging its icon onto the workspace.

   c. **Close the tool windows simultaneously by choosing Close from the Session menu.**
   All three tools should close to icons.

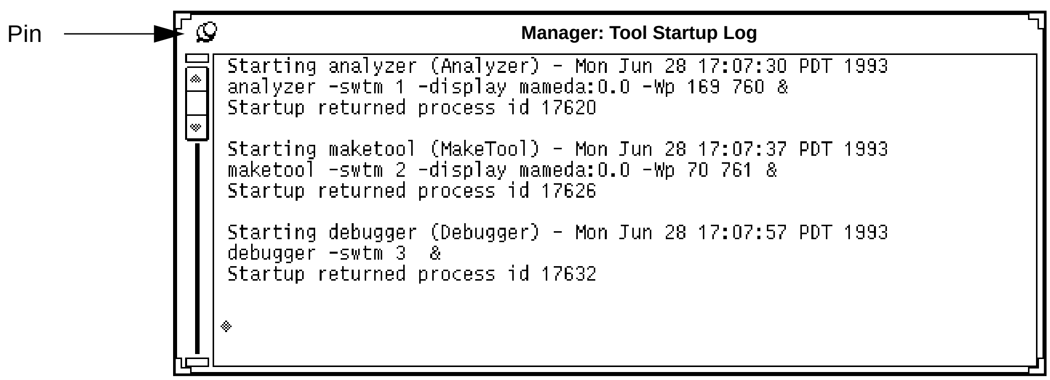
   d. **Open the tool windows again by choosing Open from the Session menu.**

All three tool windows should be displayed. Bring the Manager window into the foreground again.

If you don't want the tool windows visible while you are working but you do want them readily accessible, you can hide them. Hiding a tool means the tool window is open and ready to work in, but has been made invisible.

e. **Hide the tools by choosing Hide from the Session menu.**
All of the tool windows that are open should disappear from your screen. You can redisplay all of them simultaneously by choosing Show from the Session menu.

2. **Check to see when the tools were started.**
Choose Tool Startup Log from the View menu, which displays the tools you started from the Manager and their process IDs.



To exit the Tool Startup Log, unpin the popup window by clicking on the pin.

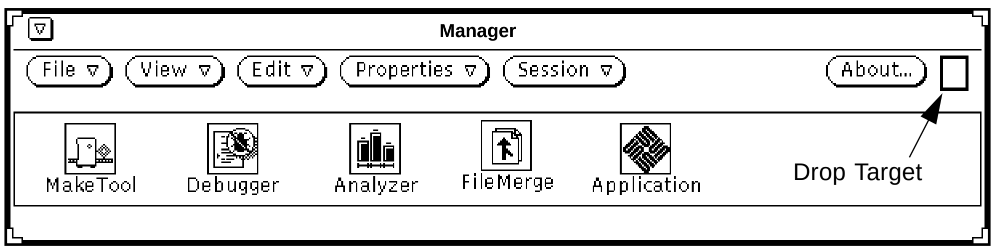
3. **Redisplay the tools by choosing Show from the Session menu.**
All three tools should reappear.

4. **Quit the tools by choosing Quit from the window menu of each tool.**
You can quit all the tools at once by quitting the Manager.

You've just learned how to manage tools from the Manager. You can continue on to the next task in this exercise or you can stop here. If you choose to stop, you can either resume the tutorial at a later time by closing the Manager to an icon, or you can exit the programming environment by quitting the Manager.

2.2.2 Starting a Non-ProWorks or Non-SPARCworks Application

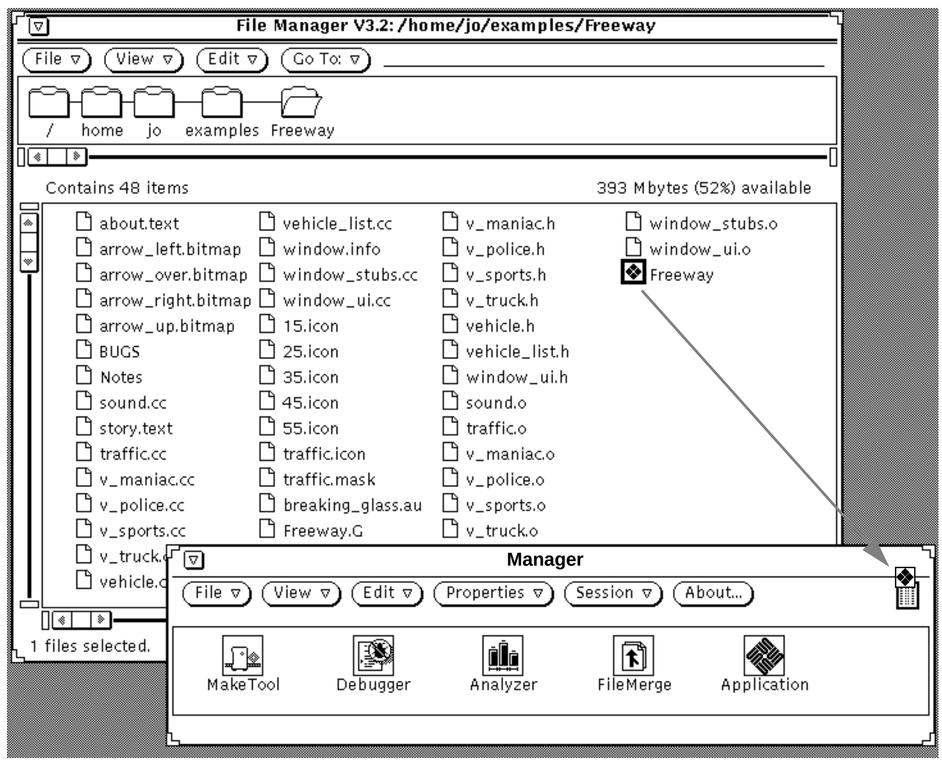
An OpenWindows Drop Target is located in the upper-right corner of the Manager window. The Drop Target is a convenient way to add an arbitrary application to the Manager from which you can start it.



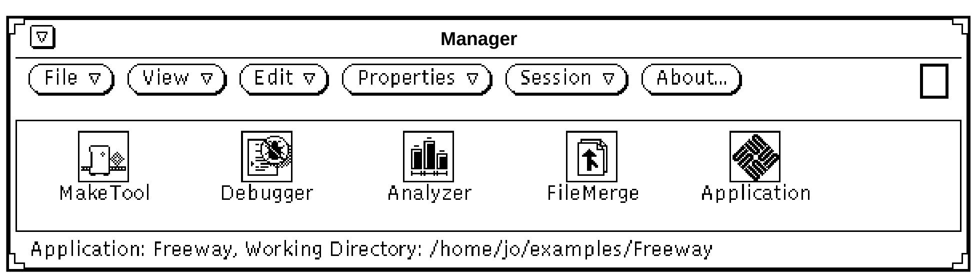
To start your own application from the Manager palette, you should have the Manager palette and the File Manager (a DeskSet™ tool) open.

1. **Select the application's icon or full path name from a command tool.**
   A sample File Manager is shown in the following figure.

2. **Drag the icon or path name onto the Drop Target.**



3. **"Drop" the icon or name by releasing the mouse button.**
This step identifies the application and associates it with the Application icon in the Manager palette. A message in the footer contains the application's name and directory.



4. Start the application by double-clicking on the Application icon or by dragging it onto the workspace.

Because session control only works with tools that communicate using the ToolTalk™ interprocess communication protocols, most applications you start this way will respond to session control commands.

2.2.3 Adding and Deleting Tools

You aren't restricted solely to the tools provided in the toolset. You can integrate your own tools or applications by adding them to the Manager palette. For instance you might want to add your preferred editor or file management tool to the palette for quick access.

To add a tool, you need to

- Duplicate an existing tool in the palette
- Set the properties for starting up the new tool
- Apply the changes

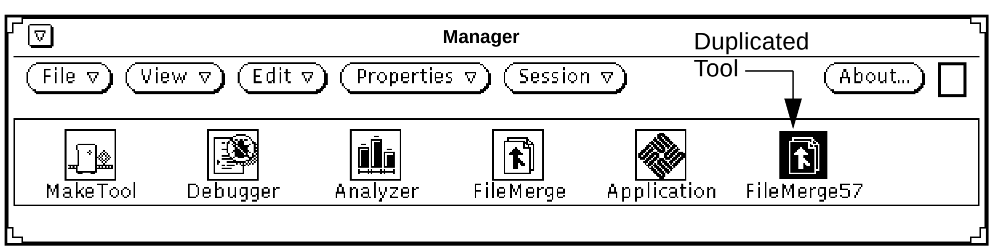
In this part of the exercise, you add the OpenWindows Command Tool to the palette.

1. Start SPARCworks and open the Manager palette (see Exercise 1, step 1).
If you left SPARCworks running from the last task, skip to the next step.

2. Add Command Tool to the Manager palette.

a. Click on any tool in the palette.

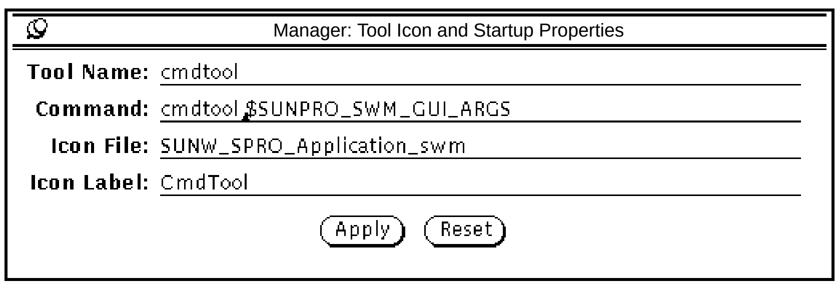
b. Choose Duplicate Tool from the Edit menu.
An icon representing the duplicated tool, indicated by highlighting, will appear in the palette (in this example, FileMerge is duplicated):



The duplicated tool has the same functionality as the original. Duplicating a tool gives you the ability to work on more than one program at a time in the same session.

c. **Click on the duplicated tool icon to select it and then choose Selected Tool from the Properties menu.**
When the Tool Properties window opens, you see that the properties are set for the selected tool. Since FileMerge was duplicated in these examples, the Tool Properties window shows the settings for FileMerge.

d. **Edit the properties for the Command Tool.**
Modify the text fields as shown in the following figure:



Tool Name — Enter the name of the tool being added. Each tool that you add should have a unique identifier; that is a tool name different from other tools in the palette.

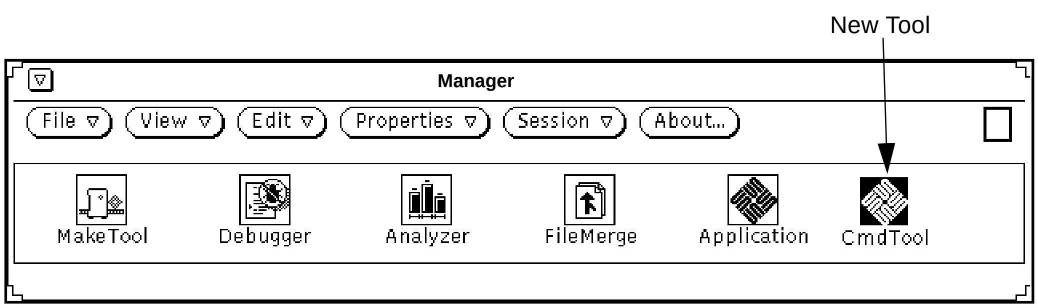
Command — Enter the command used to start the tool and the argument `$SUNPRO_SWM_GUI_ARGS`, which specifies the initial position of windows when drag-and-drop is used. For other special line arguments, see *Managing the Toolset*.

Icon File — Enter the file name of an existing icon glyph that is currently unused. Use `SUNW_SPRO_Application_swm` for this exercise.

Icon Label — Enter the name you want the icon in the Manager palette to have.

e. **Establish the properties by clicking on the Apply button.**
Click Reset to change the property values back to their last saved settings.

f. Start the new tool.
Start the new tool by either double-clicking on the icon or dragging the icon onto the workspace.



If a new tool icon does not appear or if the new tool doesn't start, check that the entries in the text fields of the tool properties window are correct.

3. **Delete a tool from the palette.**
For this task, you delete the Command Tool you just added.

   a. **Select the CmdTool icon.**

   b. **Click on the Edit button or choose Delete Tool from the Edit menu (Delete Tool is the default Edit command).**
   When the selected tool is deleted, its icon is removed from the palette. If it remains in the palette, you were unsuccessful in deleting the tool and should repeat steps a and b.

   You can always restore a tool that you deleted during a programming session by clicking on Restore *toolname* in the Edit menu. Every time you delete a tool, a restore command for that tool is automatically appended to the bottom of the Edit menu.

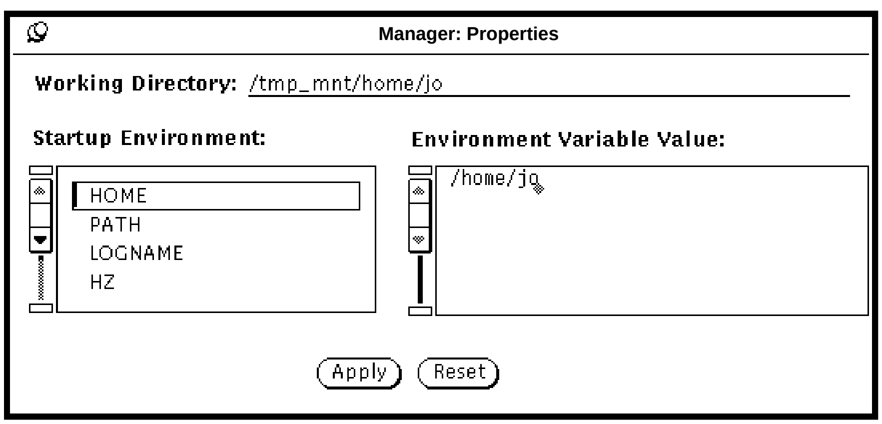
Now that you know how to add, delete, and restore tools in the Manager palette, try adding one of your own tools or the editor you prefer.

2.2.4 Customizing the Manager

Let's say that you have two projects to work on and each project requires unique modifications to the Manager. For instance, one project could require a tool, such as an editor, while the other project might require you to use another tool, such as a navigator.

You can create two Managers with customized toolsets and have their working directories set appropriately for each project. This allows you to go from one programming session to the other without having to change directories or call up an editor or navigator each time you switch projects.

1. **Change the Manager directory.**
   Choose All Tools from the Properties menu to open the Manager Properties window. Try changing the directory of the Manager.



Working Directory — Directory in which the tools are started. To change the directory, edit the path name. The default directory is the directory in which you started the Manager.

Startup Environment — List of environment variables. To change the value of a variable, select it and change its value in the Environment Variable Value column.

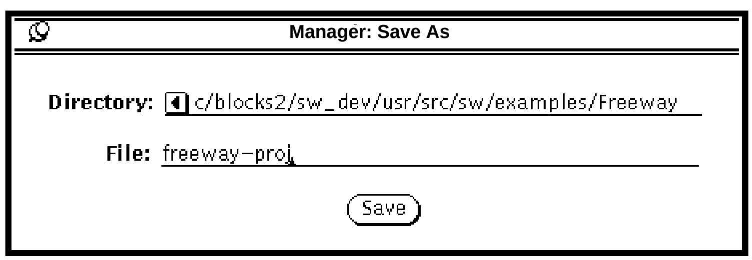
Environment Variable Value — Value of currently selected environment variable. For details on the environment variables, see *Managing the Toolset* and the operating system documentation for the `sh` and `setenv` commands.

Click on Apply to establish the changes, or click on Reset to change the properties to their last applied settings.

New tools that you start after the changes are applied inherit the new variable values; however, tools that were started before the modifications were made don't inherit the new values.

When you save the new Manager properties, you actually save the properties of each tool, not the working directory or the startup environment. Changes to the working directory and the startup environment are only in effect during the current session. To permanently save Manager customizations, save the modified Manager to a new file (see the following step).

2. **Save the new configuration by choosing Save As from the File menu.**



   a. **Put the customized Manager configuration file into the desired directory.**
   b. **Give the customized Manager a file name.**
   c. **Save the changes by clicking on the Save button.**

3. **Start the customized Manager.**
   At the shell prompt, type the appropriate command as shown, followed by the path name of the customized Manager:

   For ProWorks: `proworks /`*path*`/freeway-proj &`

For SPARCworks: `sparcworks /`*path*`/freeway-proj &`

If you have more than one customized Manager, you can merge the tool icons each contains into one palette by including each customized Manager's file name after the command. For example, if you have two customized Managers called "A" and "B" respectively, you can merge the icons into a single palette by typing `A B` after the command:

For ProWorks: `proworks A B &`

For SPARCworks: `sparcworks A B &`

4. **Quit the Manager.**

You are done with Exercise 2. You now know how to add tools to the Manager palette and delete them, control a programming session, set tool properties from the Manager, and change the working directory of the Manager.

In the next exercise, you gain experience running the rest of the tools using the Freeway sample program.

2.3 Exercise 3 — Building a Freeway

In this exercise, you lay the groundwork for the rest of the tutorial by building the Freeway program. With MakeTool, you can do more than just build a program with a makefile. You can open the Makefile Browser to interpret makefile statements by expanding the *rules* (targets, their dependencies, and the methods used to build the targets) and macros in the statements.

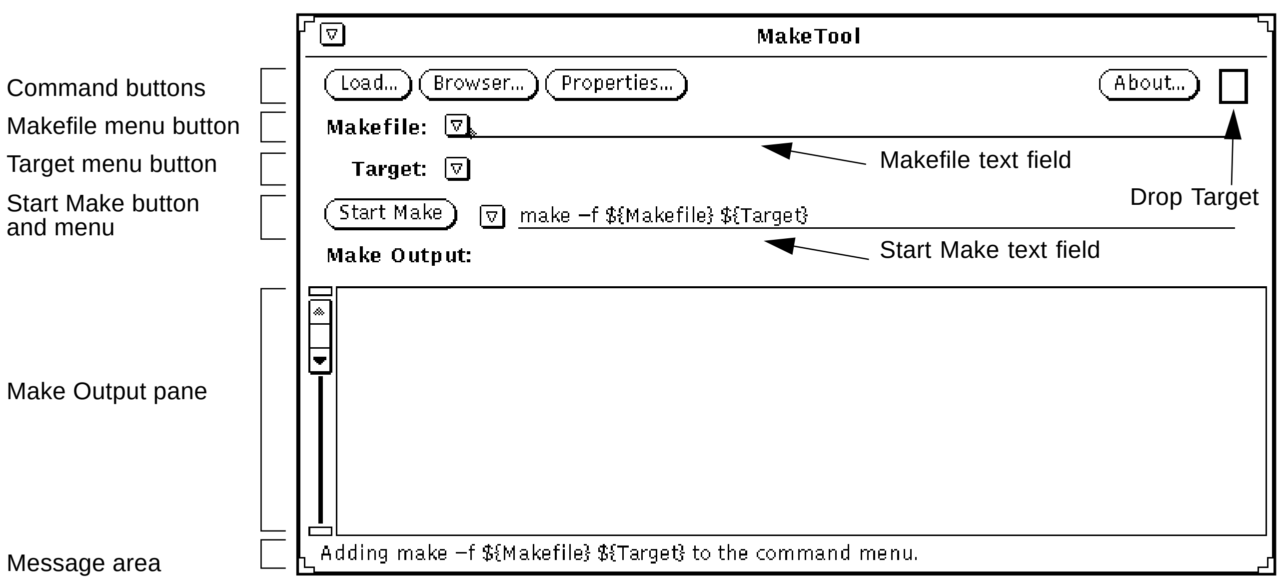
Goals — In this exercise you will learn how to

- Start MakeTool
- Load a makefile
- Do a build
- Browse makefile statements

2.3.1 Running `make`

This part of the exercise shows you how to load and run a makefile.

1. **Open MakeTool from the Manager by double-clicking on the MakeTool icon.**
   The MakeTool window appears with the default `make` command on the Start Make text field.



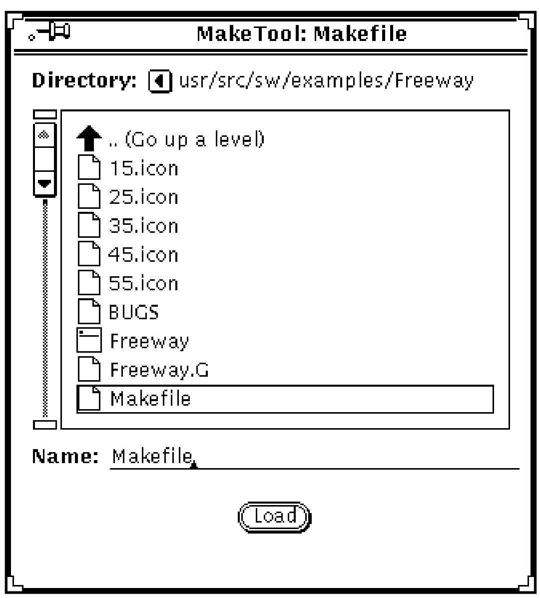
You can also start MakeTool or any other tool in the palette by dragging its icon onto the workspace.

2. **Load the Freeway makefile**
   If the Manager's working directory is the directory containing Freeway, MakeTool will automatically find and load the makefile. If the working directory does not contain a makefile or you are in the wrong directory, then you should follow steps a and b.

   a. **Click on the Load button.**

   b. **In the Makefile window, change to the directory containing the Freeway sample program and double-click on the Makefile icon.**
      You can also select `Makefile`, and click on the Load button.

      The sample program should be in one of the following locations:
      /*your_directory*/`SUNWspro/SW3.0.1/examples/Freeway` or
      `/opt/SUNWspro/SW3.0.1/examples/Freeway`

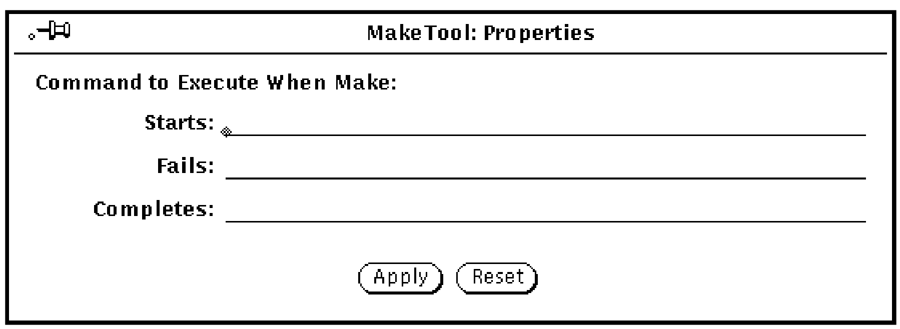


When the makefile has been loaded, the makefile path will appear in the Makefile text field.

3. **Click on the Target menu button in the MakeTool window and choose the debug option.**
   This menu contains all the top-level target files for the makefile. Choose `debug` to build an executable that can be debugged.

4. **Have MakeTool notify you when the build is completed.**
   Before doing the build, set up the MakeTool properties to inform you whether it completes or fails. The MakeTool Properties window allows you to write commands that are executed at the beginning or the end of a build.

   a. **Open the MakeTool Properties window by clicking on Properties.**
   If others are waiting on your build or you want to work on another project while the build is in progress, you can create notification messages that will be sent to others or to yourself when the build starts, completes, or fails.



   Starts — Command that is executed when a build has begun.

   Fails — Command that is executed when a build has failed or aborted.

   Completes — Command that is executed when a build has completed.

   b. **Enter the following commands in the Starts, Fails, and Completes text fields:**

   `mail` *user* `< $HOME/make_begun`

   `mail` *user* `< $HOME/make_fail_message`

   `mail` *user* `< $HOME/make_complete_message`

   In this example, we created the files `make_begun`, `make_fail_message`, and `make_complete_message`. You can choose your own names for the files you create.

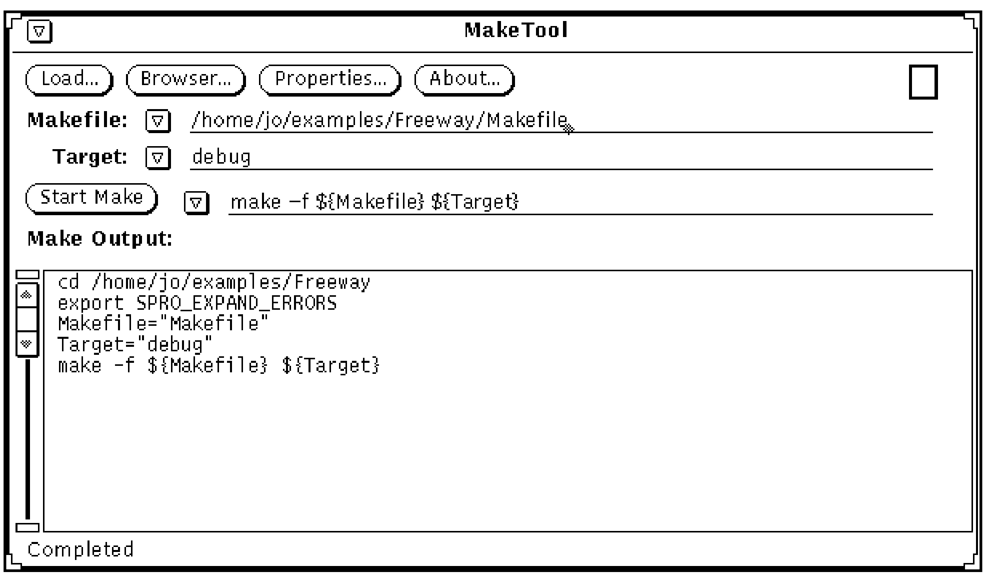
Note – MakeTool executes Bourne shell commands; therefore, you must type in full path names in the properties window since the tilde (~) symbol is not recognized.

c. **Set the commands by clicking on Apply.**

5. **Run** `make` **by clicking on the Start Make button (which toggles to Stop Make).**
   MakeTool appears with a default `make` command on the Start Make text field. For this exercise, use the default command.

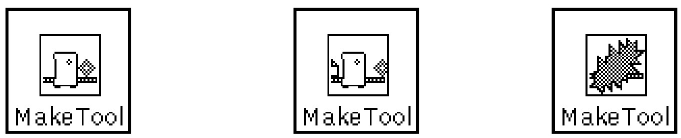
   If you need to stop the build in progress, click on the Stop Make button. A message appears in the message area of the MakeTool window notifying you that the build has been aborted.

   You can see the transcript of the build in the Make Output pane. The pane provides a scrollable read-only display of the `make` command's output.



6. **Watch the build progress through MakeTool's animated icons.**
   MakeTool in its closed form is represented by one of three icons. Each icon represents a different condition of a `make` build process:

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



Successful Make   Make in Progress   Failed Make

2.3.2 Issuing Commands from MakeTool

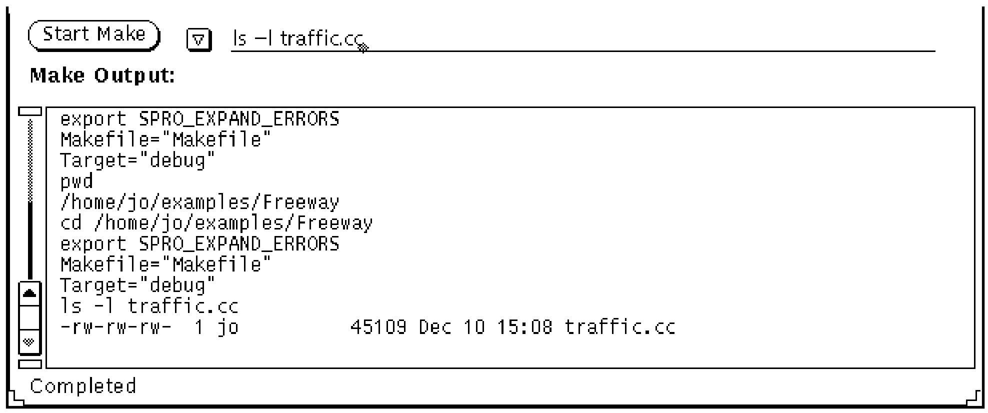
You can do more than just run a `make` command in MakeTool. For instance, you can compile files, perform certain UNIX commands (for instance, do an `ls -l` on a Freeway file) or modify the default `make` command. MakeTool maintains a command history of all the commands you invoke in a session, so if you want to invoke a previously issued command without having to retype it, you can select it from the command history menu and click on Start Make.

1. **Show the working directory and then get a file listing for the file** `traffic.cc`**.**

   a. **Go to the Start Make text field and delete the default** `make` **command.**

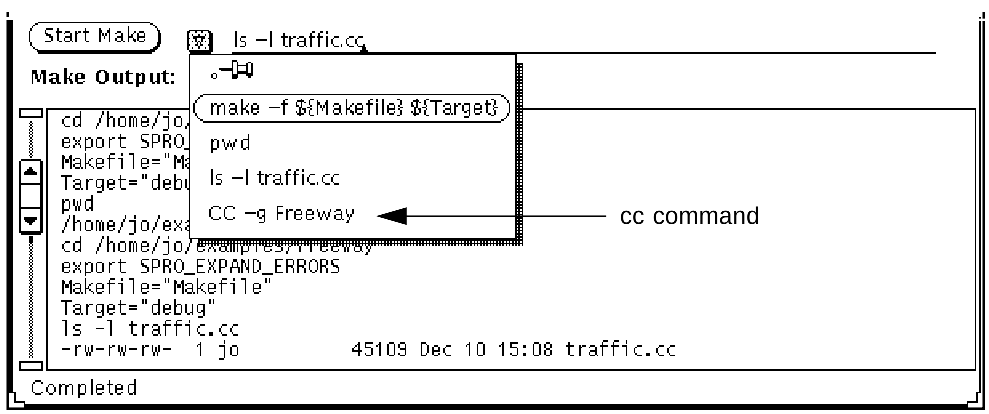
   b. **Type** `pwd` **to see the working directory and press Return.**
   The Make OutPut pane displays the command followed by the directory name.

   c. **Delete** `pwd` **from the text field and type** `ls -l traffic.cc` **and press Return.**



2. **Check the command history by clicking MENU on the menu button next to the Start Make button.**
   You'll see a list of the commands that you've issued.

Note – In the following figure, a compilation command is shown in the menu. Your Freeway program has been compiled; the command is here only to show that you can invoke CC commands from MakeTool.

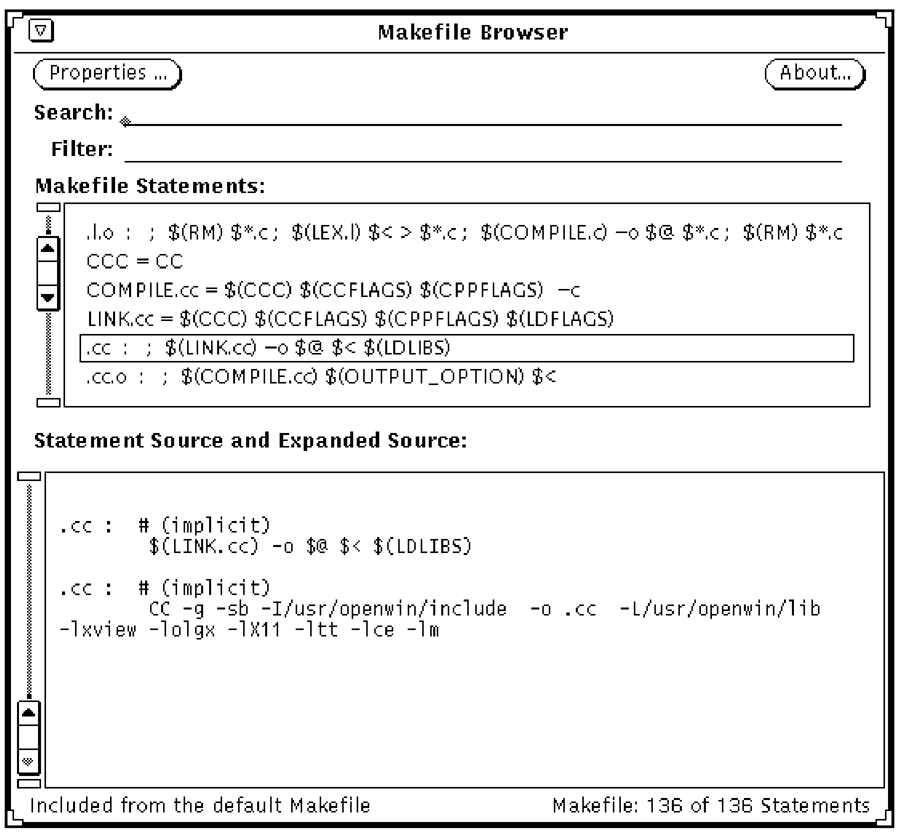


For more information on the Start Make text field, see *Building Programs with MakeTool*.

2.3.3 Interpreting Makefiles

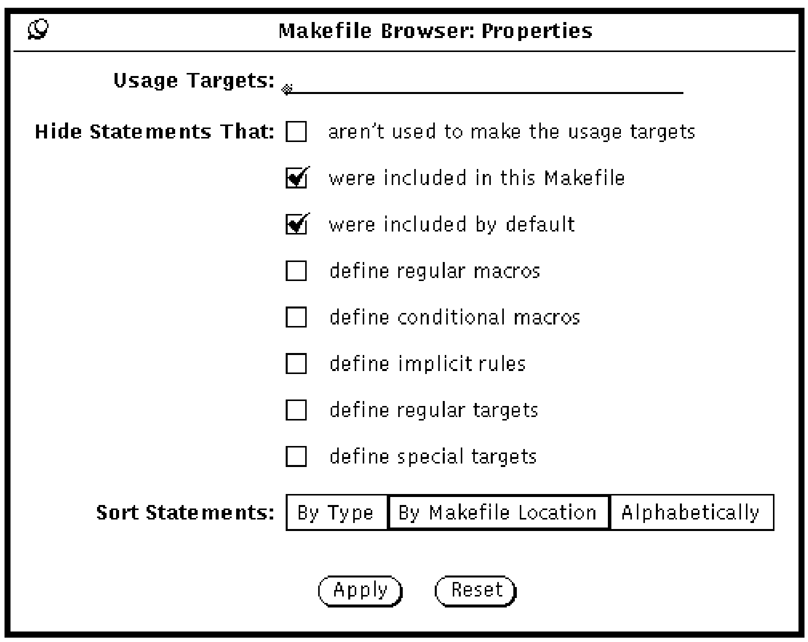
Now that you've built Freeway, examine some of the makefile statements in the Makefile Browser window.

1. **Open the Makefile Browser window by clicking on the Browser button.**
   The Makefile Browser window displays Freeway's makefile statements. The upper pane displays the statements as they appear in the makefile. The lower pane displays a single statement and its expanded form. At the bottom of the window is a message area that tells you how many statements the file contains.



2. **Examine some makefile statements.**
   Rather than scrolling through the makefile for particular statements, narrow the search by filtering out the statements that were included by default and that were included in the makefile.

   a. **Open the Makefile Browser Properties window.**
      Click on the Properties button to open the Makefile Browser Properties window.



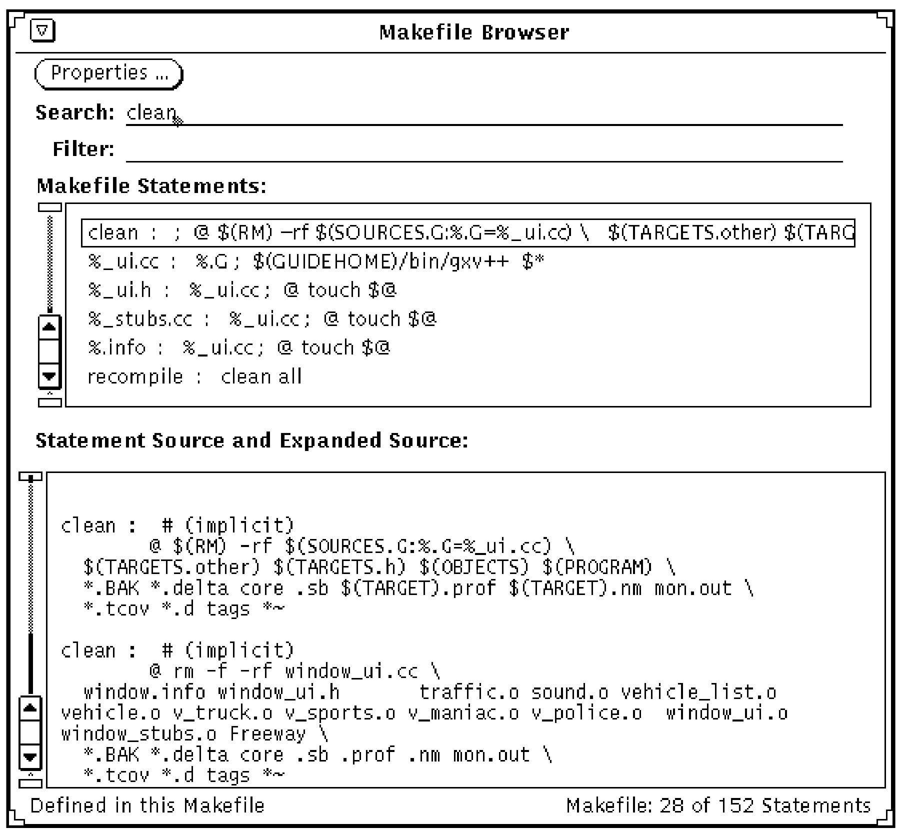
   b. **Hide statements included by default and included in the Freeway makefile.**
      Click on the appropriate boxes in the Hide Statements That list as shown in the preceding figure.

      Check that the default sort statement, By Makefile Location, is selected and click on Apply.

      In the next figure, the message area at the bottom of the window tells how many makefile statements out of the total number of statements are now displayed.

Note that you can sort statements by type, file location, or alphabetically. For more information on Makefile Browser properties, see *Building Programs with MakeTool*.

c. **Search for `clean`.**
Go back to the Makefile Browser window. In the Search text field, type in `clean` and press Return.

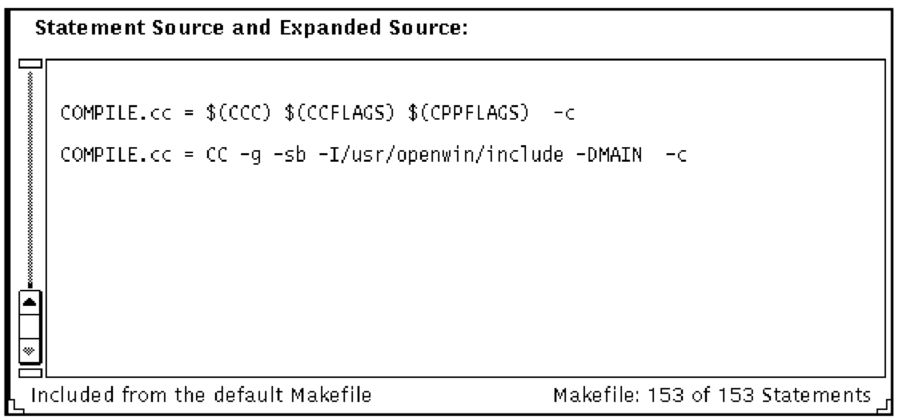


The Browser searches for the first occurrence of `clean` in the makefile within the constraints set in the Properties window. To move to the next occurrence of `clean`, press the down-arrow key on your keyboard. You can continue to search for all occurrences this way. When the last one is found, the search wraps back to the top. When you're done, deselect the hide statements in the Properties window and click Apply.

d. Expand a CCFLAGS statement.

Search for CCFLAGS just as you did for clean. Click on the first occurrence of it in the Makefile Statements window.

You see the statement repeated in the Statement Source and Expanded Source pane with its expanded form immediately below it.



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

For detailed information on the Makefile Browser window and on searching and filtering statements, see *Building Programs with MakeTool*.

3. Quit MakeTool by choosing Quit from the menu button.

You are done with Exercise 3. As a self-exercise, write your own makefile and load it. You can then modify the make command again or set other properties in the Makefile Properties window and search for statements.

In Exercise 4, you learn to use the Debugger by checking for runtime errors in Freeway, setting breakpoints, navigating the stack frame, and creating a command button.

2.4 Exercise 4 — Debugging Freeway

The Debugger is a GUI extension of the source-level debugger, dbx. You can invoke any dbx command from the Debugger window. This exercise only demonstrates a few of the Debugger's features, just enough to get you familiar with the Debugger and its GUI interface.

You can do more than just debug a program with the Debugger. A program that is bug free can still be inefficient and slow. The second part of this exercise will show you how the Debugger works in conjunction with the Analyzer to help you improve program performance.

Goals — In this exercise, you learn how to

- Run a program from the Debugger
- Set breakpoints
- Inspect data
- Examine stack frames
- Create your own command button
- Perform runtime checking

2.4.1 Running Freeway

Now that you've built the Freeway program, you can run it from the Debugger.

1. **Start the Debugger.**
   Double-click on the Debugger icon in the Manager palette. Then click on the Program menu button to open the Program Loader window to load Freeway.

   You also can start the Debugger in one of the following ways:

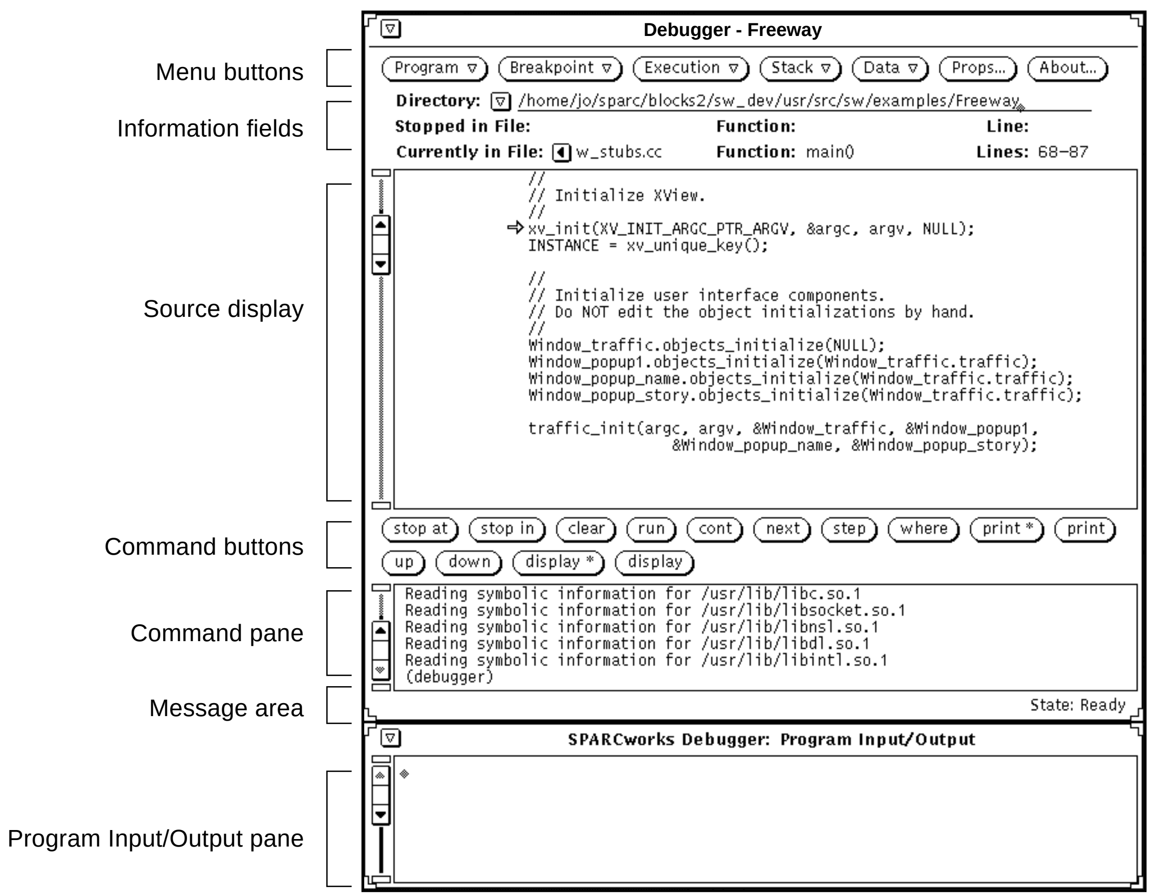
   - Type the following command at a shell prompt:

     ```
     debugger Freeway &
     ```

     When you supply a file name argument, the Debugger starts up and loads the specified file, in this case, `Freeway`.
   - Start the Debugger from MakeTool by entering the `debugger` command on the Start Make text field in MakeTool:

When the Debugger starts, two windows appear: the Debugger window and the Program Input/Output window.

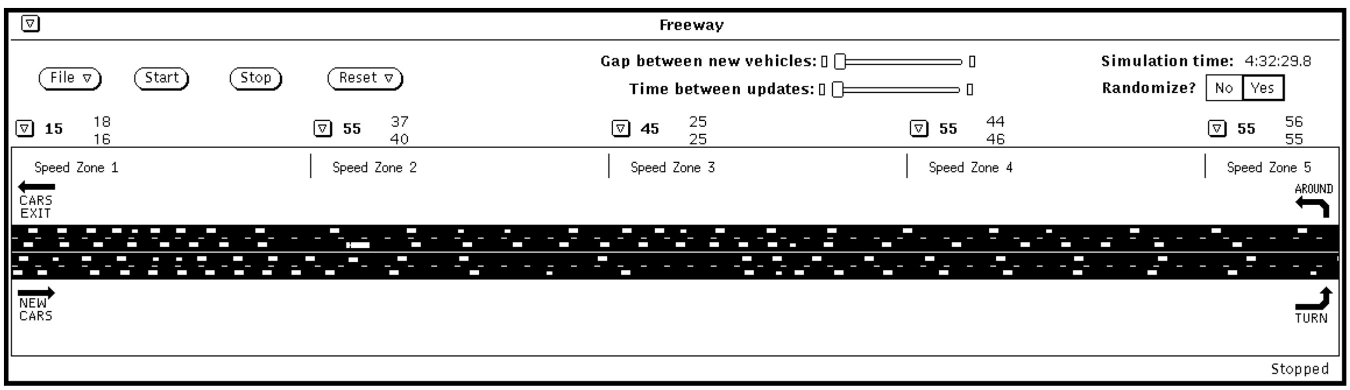
When the program is loaded, the source display shows a source file and the Debugger prompt will be displayed in the command pane.



For Debugger start up options for help, remote display, source display buffer, window applications, and debugging options, see *Debugging a Program*.

2. **Run the Freeway program.**
   Once the Freeway program is loaded, try running it to see what the program does. To run the program, click on the run command button.

   You can also choose Run from the Execution menu or type `run` at the Debugger prompt. Freeway's main window will appear on your screen:



3. **Start Freeway.**
   To start the Freeway, click on the Start button in the Freeway window. Try reducing and increasing the speed in some zones, and the distance between vehicles.

   You change the attributes for a vehicle on the freeway by selecting it. A Vehicle Information (VI) window appears in which you can change the type of vehicle, its state of movement, its current speed, and its maximum speed. To apply the new attributes, simply unpin the VI window.

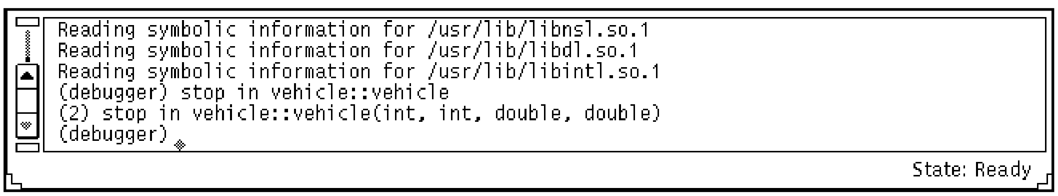
4. **Stop the program by clicking on the Stop button.**
   At this point, you can either close the window to an icon if you want to play with Freeway later; or quit the application and continue with the next task.

2.4.2 Setting Breakpoints

The Debugger enables you to set breakpoints via the Breakpoint menu, the command buttons, or the command pane.

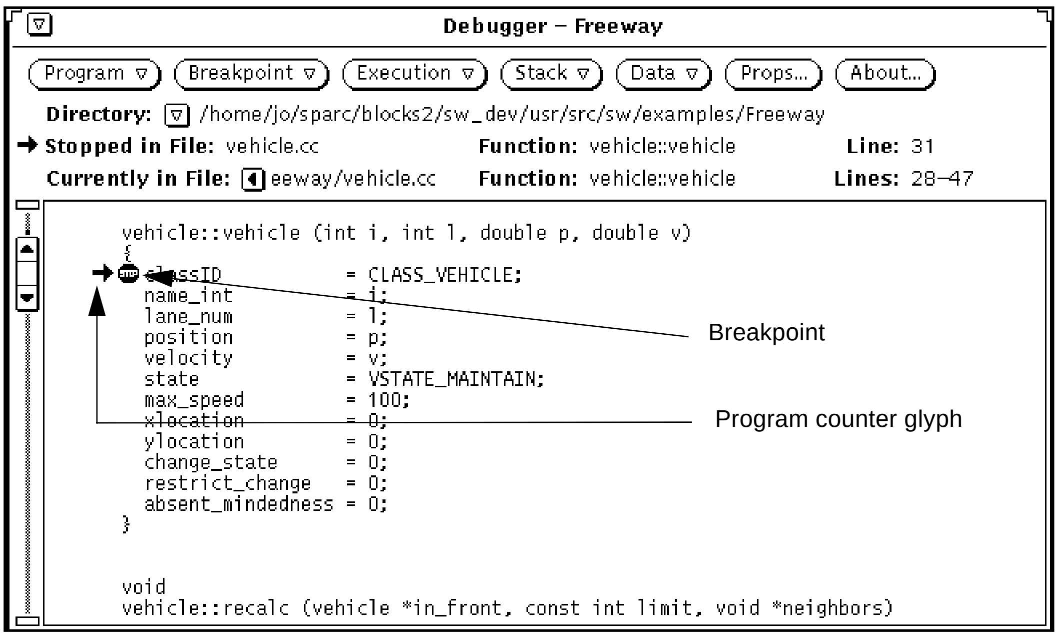
1. **Set breakpoint at** `vehicle::vehicle`**.**

   a. **Type** `stop in vehicle::vehicle` **at the Debugger prompt and press Return.**



   b. **Now run the program by clicking on the run command button (or choose Run from the Execution menu or type** `run` **at the Debugger prompt).**
   The Freeway program appears on your screen.

   c. **Start the application by clicking on the Start button in the Freeway window.**
   The Freeway window is large; you may want to move the window to the background or close it to an icon.



When the application stops at the breakpoint, the source display shows the source file that contains the function, and the name of the file appears in the Information field, as does the line number. A breakpoint, designated by a stop sign glyph, is set at the line.

2. **Clear the breakpoint.**
Select the line containing the breakpoint (put the cursor on the line and double-click), then click on the clear command button. The breakpoint is removed while the program counter remains at the line where the application is stopped.

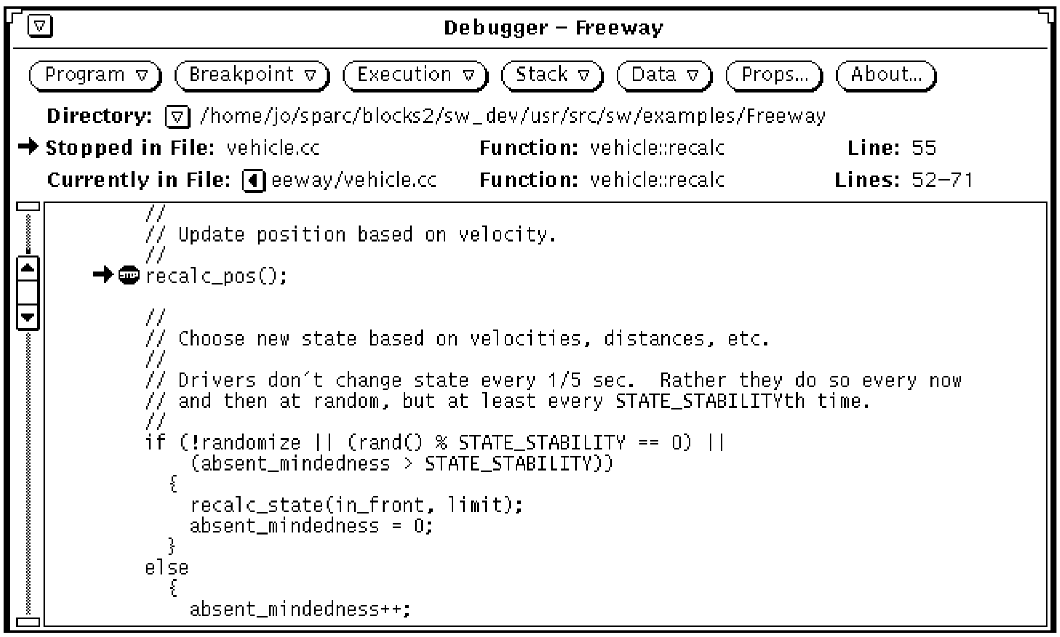
   You can also select the line containing the breakpoint and choose Clear from the Breakpoint menu.

3. **Set a breakpoint on the line containing the function** `vehicle::recalc`**.**
Look in the source display, this line should appear at the bottom of the window. If it does not appear, scroll down a few lines.

   a. **Select the function.**

b. **Click on the stop at command button and then click on the** `cont` **command button.**

A breakpoint is placed on line 55 at the call to `recalc_pos`. It is placed there because stop at always goes on an executable line. Line 55 is the first executable line in `vehicle::recalc`.



Stop At is also available from the Breakpoint menu and Continue is available from the Execution menu.

4. **Set breakpoints in the functions** `traffic_generate` **and** `traffic_simulate`**.**

Type `stop in traffic_simulate` at the Debugger prompt and press Return. Wait for output confirming that the breakpoint has been set:

```
(debugger)stop in traffic_simulate
(4) stop in traffic_simulate(unsigned long, int)
```

Then type `stop in traffic_generate` and press Return.

Remember, if you set a breakpoint at the wrong function, you can delete it by selecting the line containing the erroneous breakpoint and clicking on Clear in the Breakpoint menu.

5. **Continue the application and watch the cars entering the freeway.**
   Click on the `cont` button twice (or choose Continue from the Execution menu) to see the breakpoints you've set. Click on `cont` a few more times; as the vehicles progress, you can watch the simulation timer advance.

6. **Clear all the breakpoints at once.**
   Choose Clear All Breakpoints from the Breakpoint menu to remove all the breakpoints that you've set. The message `delete all` will appear in the command pane.

For more information on breakpoints and stepping through code, see *Debugging a Program*. For a quick summary of Debugger or dbx commands, you can also type at the Debugger prompt:

`(debugger) help` <*command-name*>

2.4.3 Inspecting Variables

Solaris

The Visual Data Inspector (VDI) is available only on platforms running Solaris 2.x. If you are running Solaris 1.x, skip this part of the exercise and continue with "Displaying Data" on page 2-42.

The VDI enables you to track and graphically display variables, including data structures. With the VDI, you can put the graphical representation of variables into separate buffers, minimize or expand the display of information, and follow pointers to other variables.

1. **Set a breakpoint in the function `traffic_advance`.**
   Type `stop in traffic_advance` at the Debugger prompt, press Return, and then click on the `cont` button to advance to the breakpoint.

2. **Look at a variable value with the Visual Data Inspector (VDI).**
   Choose Inspector from the Data menu.

3. **Select a variable to inspect.**
   To get to the location of the variable definition for `upperlane`, single-step through the code with `next` and `step`.

c. Click on the step command button twice.
When you invoke the `step` command, if the line that is executed contains a function call, it stops at the first line of that function.

You can also select Step from the Execution menu or by typing it in at the Debugger prompt. You can specify the number of lines of code you want to step through at a time by providing a number argument to either `step` or `next` at the Debugger prompt.

d. Type `next 5` **at the Debugger prompt and press Return.**
You should be at line 365, which shows

```
if (IsInsideLane(j)) limit += 5;
```

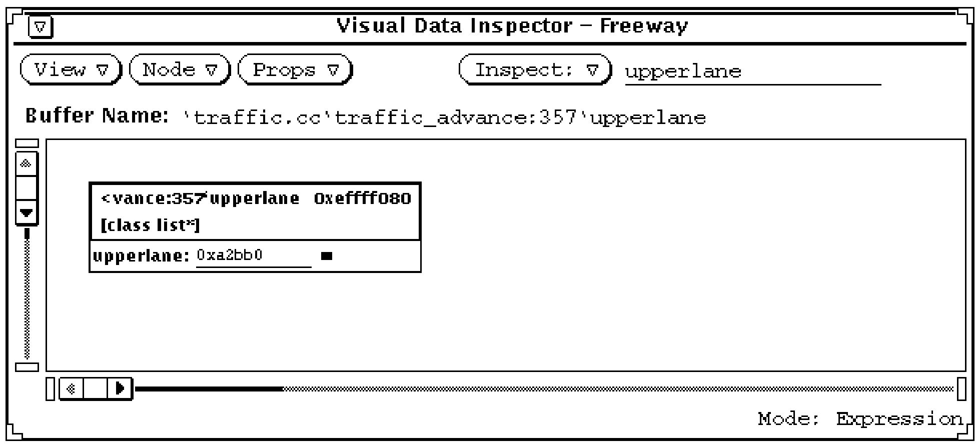
e. Click on step four more times to take you where `upperlane` **is defined.**
You should be at line 373 in `traffic.cc`, which shows

```
if (next->hasValue())
```

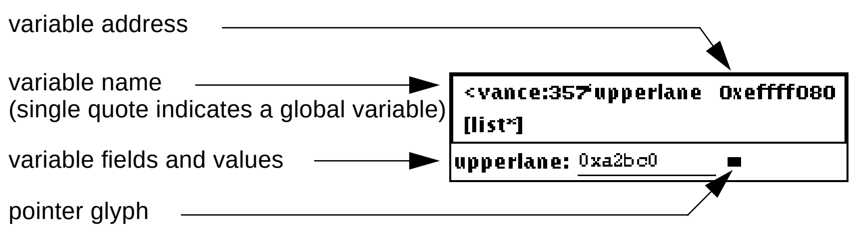
f. Select the variable `upperlane` **that is defined just before the** `if` **conditional in the source display. In the Data Inspector, click on the Inspect button to enter the name of the variable in the text input field.**
Two other ways to enter the variable names are

- Select the variable in the Source Display and drag the variable name to the text input field.
- Select the variable in the Source Display and use the Copy and Paste keys to enter the name in the text input field.

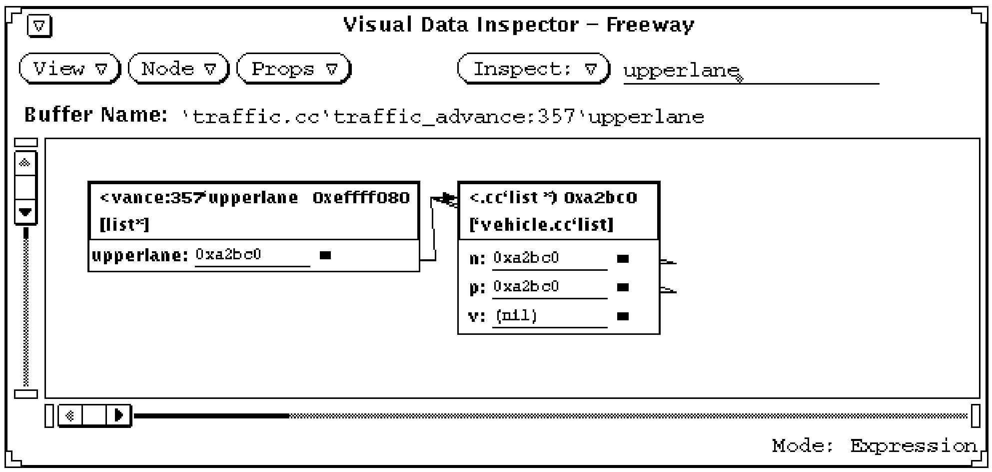
4. **Click on the Inspect button in the VDI window.**
The display pane of the VDI window shows the variable as a node:



Nodes contain the following information:



5. **Follow the pointer by clicking on the glyph.**
Every node has a pointer glyph. If the value of a parameter is non-nil, you can follow its pointer to another variable by clicking on the glyph. The shape of the cursor changes to a stopwatch indicating that inspection is in progress.



When you follow the pointer from `upperlane`, it points you to `‘vehicle.cc‘list`. Move the node so you can clearly see how the node refers back to the node for `upperlane`.

6. **Change the node displays.**
   Select one of the nodes and choose Minimize from the Node menu to close the node to an icon. If you have multiple nodes, use this menu command to free up space in the display pane.

   Select Maximize from the Node menu to expand it.

7. **Quit the VDI window.**

See *Debugging a Program* for a thorough discussion of the Visual Data Inspector.

2.4.4 Displaying Data

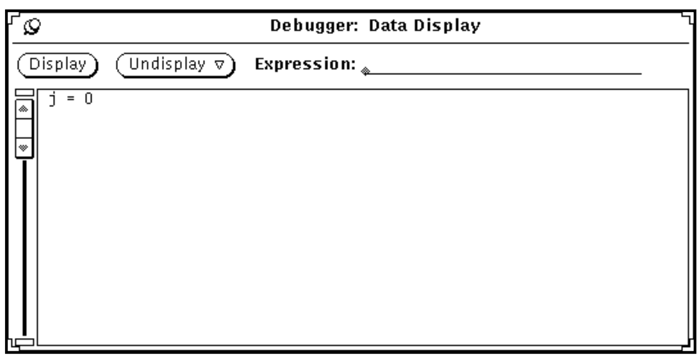
By observing the values of expressions and variables through the Data Display window, you learn how and when expressions and variables change value during the execution of a program.

1. **At the Debugger prompt, type** `stop at 259`**, press Return and then press the** `cont` **button.**
   Specifying the line number 259 sets a breakpoint at the line containing `segment_vel` in the function `traffic_stats`:

segment_vel[j][i] = 0.0;

To move to line 259 in the source display, you must press the `cont` button eight more times to cycle through the program. This portion of the code calculates the average velocities of the vehicles.

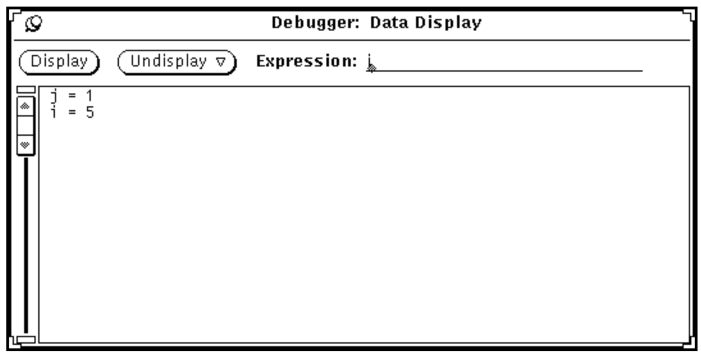
2. **Open the Data Display window by selecting the integer, `j`, and clicking on the Display button.**
   The Data Display window shows the value for `j` in the display pane:



   You can also open the Data Display window by choosing Display <expr> from the Data menu.

3. **Add `i` to the data display.**
   Type `i` in the Expression field and press Return.

4. **Step through the program and watch the value of `i` increase incrementally.**
   Click on `next` or `step` until you see the value of `j` change from `0` to `1`.



5. **Turn off the display.**
   The Debugger will display the values of the variables until you explicitly turn off the display.

   a. **Choose All Items in the Undisplay menu.**

   b. **Quit the Data Display window by unpinning the window.**

2.4.5 Moving Through the Call Stack

You can display the *call stack* (a list of all active routines) for the current process by invoking the Debugger's `backtrace` command (`backtrace` is equivalent to the `where` command.) The command pane will show all the routines that were called during the current session.

1. **Do a backtrace to see the current call stack.**
   Choose Backtrace from the Stack menu. The command pane shows that `where` (Backtrace) was invoked and prints the call stack.

```
(debugger) where
=>[1] traffic_stats(delay = 1), line 257 in "traffic.cc"
  [2] traffic_simulate(__UNNAMED_ARG__1__ = 448712, __UNNAMED_ARG__2__ = 0), line 452
in "traffic.cc"
  [3] notify_itimer(0x0, 0x0, 0x1, 0x0, 0x0, 0x0), at 0xef694b3c
  [4] ndis_default_prioritizer(0x6d8c8, 0x40, 0xeffff360, 0xeffff2e0, 0xeffff260,
0x22), at 0xef5fe654
  [5] notify_client(0xd923c, 0x0, 0xef6fc43c, 0x8a89a320, 0x0, 0x0), at 0xef625174
  [6] ndis_default_scheduler(0x0, 0xdbe88, 0x0, 0x0, 0x0, 0xffffffff), at 0xef6950fc
  [7] scheduler(0x1, 0xdbe88, 0x0, 0x0, 0x0, 0xd9218), at 0xef6c0968
  [8] ndis_dispatch(0x0, 0xef729478, 0x1000, 0x0, 0x0, 0x0), at 0xef69489c
  [9] notify_start(0xef7292cc, 0xef722b78, 0x11, 0x0, 0x0, 0xeffff8a8), at 0xef691d9c
  [10] xv_main_loop(0x6d8c8, 0xeffffa54, 0x52028, 0x5209c, 0x64a70, 0x6d928), at
0xef6ef9f0
  [11] main(argc = 1, argv = 0xeffffa54), line 91 in "window_stubs.cc"
(debugger) down
Already at the bottom call level
(debugger) up
Current function is traffic_simulate
(debugger)
```

2. **Move through the stack by clicking on the up command button and then the down command button.**
   Note the change in the source display as you move through the stack. When you move up a frame to visit the next function, you see a hollow arrow in the source display that points to the function.

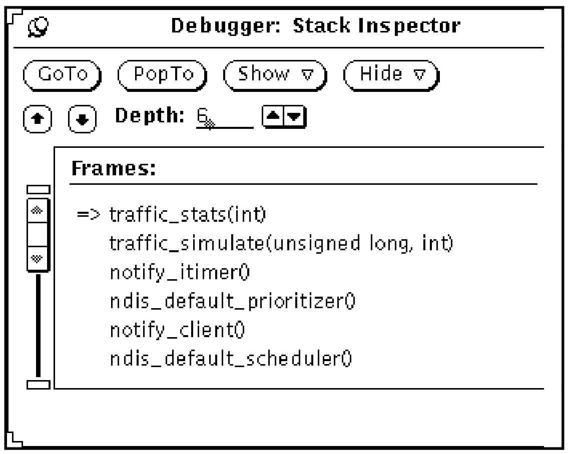
   To redisplay the function that the program is currently stopped at, move the cursor over the solid arrow pointing to the Stopped in File field (in the Information Fields area of the Debugger window) and click.

   Up and Down are also available from the Stack menu. For more information on up and down commands, see *Debugging a Program*.

2.4.6 Examining the Call Stack

Besides being able to examine the call stack, you can use the Debugger's Stack Inspector to display the stack and to selectively view the functions you're interested in.

1. **Bring up the Stack Inspector.**
   Click on the Stack menu button in the Debugger.



The Stack Inspector window shows `traffic_stats` (which was the most recently called function) at the top of the stack.

Six is the default number of routines displayed in the Stack Inspector. You can increase or decrease the number of routines displayed in the window by clicking on the up or down buttons next to the Depth field. If you increase the number of routines shown, you'll need to resize the window to see all of them at once; otherwise, you can use the scrollbar to move up and down the list. See Chapter 1, "Introduction," for information on resizing windows.

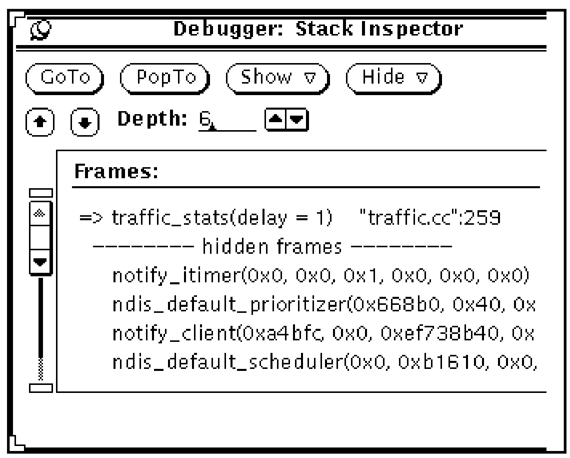
2. **Display `traffic_simulate` in the Source Display.**
   Select `traffic_simulate` in the Stack Inspector window and click on GoTo. A hollow arrow in the Source Display points you to the selected function.

3. **Go back up the stack.**
   Click on the up arrow button. You return to the current routine in the Source Display.

   You can click on the up and down arrows to move one frame at a time, or click on GoTo to move directly to the routine of interest.

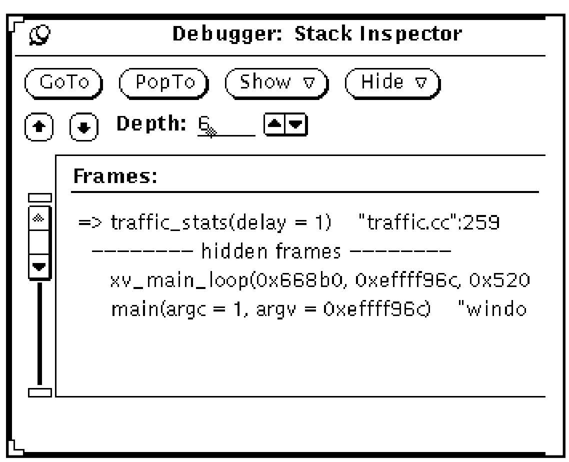
4. **Show the arguments to the functions in the Stack Inspector.**
   Choose Arguments from the Show menu in the Stack Inspector window. The arguments to the functions are displayed. To remove the arguments, click on No Arguments in the Show menu (Arguments toggles to No Arguments).

5. Filter the routines in the stack.
Select `traffic_simulate` in the Stack Inspector and then choose Hide Function from the Hide menu.



A hidden frames message appears in the Stack Inspector indicating a hidden entry.

6. Hide functions in the library containing `notify_client`.
Select `notify_client` in the Stack Inspector and choose Hide Library from the Hide menu.



All functions in the library that contains the selected function are hidden.

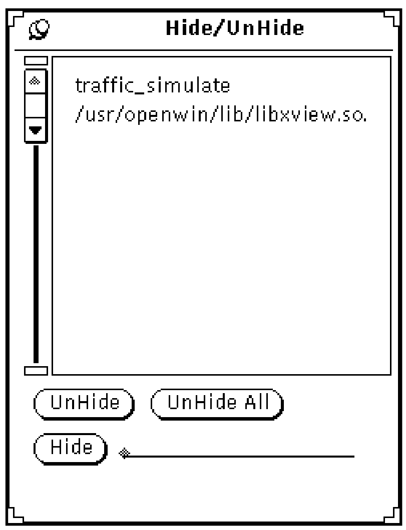
Routines at the bottom of the stack are now visible in the window.

You might find it helpful to use the Stack Inspector to hide window toolkit library calls.

7. **Unhide `traffic_simulate`.**
   Redisplay `traffic_simulate` in the stack.

   a. **Choose Hide from the Hide menu to open the Hide/Unhide window.**

   b. **Select `traffic_simulate` in the entries list and click on the Unhide button at the bottom of the window.**



Typing an expression in the Hide text field adds that expression to the list of entries in the Hide/Unhide window. The entries are hidden from view in the Stack Inspector.

You can hide all expressions in the stack that begin with a specific string by typing the string with an asterisk (for example, `traffic*`) in the Hide text field and adding it to the Hide/Unhide list by clicking on Hide.

8. **Show the hidden library functions by choosing Hidden from the Show menu.**
   Hidden shows all functions that were hidden by the `hide` commands. Clicking on Hidden again toggles the display back to its hidden frames state.

   To permanently redisplay all the hidden functions, choose Unhide All from the Hide/Unhide window.

For more information on the Stack Inspector and using the `backtrace` command to examine core files, see *Debugging a Program*.

2.4.7 Customizing the Debugger

You can customize the Debugger to fit your needs by adding or removing command buttons or by resetting the Debugger properties. This task shows you how to create your own command button and how to resize the source display pane.

2.4.7.1 Creating Your Own Command Button

You can add command buttons to the Debugger window. The window expands to accommodate the new buttons. Buttons that you create during a debugging session are discarded when you quit the Debugger. To retain the buttons, you need to put the commands that create the buttons in the `.dbxrc` (for Korn shell users) or `.dbxinit` file.

1. **Create a button called Line that displays the line number of the selected line of source code.**
   The button replaces the current action of placing the cursor over the arrow by the Stopped in File field and clicking SELECT.

   a. **At the Debugger prompt, type the following commands:**

   ```
   (debugger)kalias Line=:
   (debugger)button lineno Line
   ```

   A new command button labeled Line appears at the end of the second row of command buttons.

   b. **Display `traffic_simulate` in the source display.**
   Type `stop in traffic_simulate` in the command pane, press Return, and click `cont` to display the file in the Source Display.

   c. **Select the following line in the Debugger's source display and click on the Line button.**

   ```
   traffic_simulate(Notify_client, int)
   ```

   The command pane returns the source file's path name and line number:

```
(debugger) stop in traffic_simulate
(4) stop in traffic_simulate(unsigned long, int)
(debugger) cont
(debugger) Line
"/home/jo/sparc/blocks2/sw_dev/usr/src/sw/examples/Freeway/traffic.cc":445
(debugger)
```

See the section “Adding Buttons” in *Debugging a Program* for more information on creating command buttons.

2.4.7.2 Changing the Size of the Source Display

You can change the properties of the Debugger; the changes you make remain in effect for the current session. Once you exit the Debugger, the default properties are reset.

1. **Click on the Props button.**
2. **Choose Window Configurations from the Category menu.**
3. **Change the source display to 30 lines and the command display to 10 lines.**
4. **Click Apply.**
   You can see that both panes have increased in length.

2.4.8 Looking for Runtime Errors

The Debugger’s runtime checking feature (RTC) enables you to automatically find runtime errors in an application during development. With RTC, you can look for memory access errors and memory leaks.

A *memory leak* is a block of memory that has not been freed and is no longer accessible. The program no longer retains any references to the block. RTC finds memory leaks by looking through the entire program for pointers to heap blocks. Every time it finds a pointer, the block is marked as being referenced. RTC then goes through the list of heap blocks and reports each unmarked block as a memory leak.

1. **Look for runtime errors.**
   Examine the application for any runtime errors that might exist. If you are continuing from the last step, clear all the breakpoints in the program.

   If you quit the Debugger, you must restart it and load Freeway. If you include the `-C` switch with the `debugger` command, the `librtc` library is automatically loaded, enabling runtime checking to be turned on as soon as the Debugger starts. Otherwise, the library won’t be loaded until you run the application.

a. **Choose Run Time Checking from the Execution menu.**

b. **Choose Leaks Only from the Error Checking submenu.**
Wait for a message in the command pane notifying you that checking is turned on.

```
Reading symbolic information for /usr/lib/libnsl.so.1
Reading symbolic information for /usr/lib/libdl.so.1
(debugger) uncheck -access;check -leaks
access checking - OFF
leaks checking - ON
(debugger)
```

2. **Run the application for one cycle and then quit it.**
When the Freeway window appears, start the application. Stop it when the first vehicle makes one complete revolution, then quit the application.

```
Checking for memory leaks...
Memory Leak (mel):
Found 35 leaked blocks with total size 420 bytes
At time of each allocation, the call stack was:
        [1] operator new() at 0x309b0
        [2] list::prepend() at line 62 in "vehicle_list.cc"

Leak Summary:
        actual leaks:        35  total size:     420 bytes
        possible leaks:       0  total size:       0 bytes
        blocks in use:     1395  total size:  127277 bytes
```

Note that the error message tells you that a leak is occurring in line 62 in the file `vehicle_list.cc`.

3. **Display `vehicle_list.cc` in the source display and look for potential problem areas in the code.**
Type `file vehicle_list.cc` in the command pane. Scroll through the source display and look for the function `list::prepend`. This is the location in the code where the leaks are being detected. This is not the piece of the code causing the leaks.

While examining the file, you will notice that a line of code in the definition of the function `list::remove` has been commented out. Let's assume this could potentially be the cause of the leaks. To pinpoint the cause, set breakpoints around the suspected area and use the `showleaks` command to locate the leaks.

4. **Set a breakpoint before the suspected location in the code causing the leak errors.**
   In this example, set a breakpoint in the code as shown in the following figure by selecting the line and clicking on the stop at button:

```
void list::remove(vehicleP veh)
{
  list *i = this->find(veh);

  if (i->hasValue())
    {
      i->p->n = i->n;
      i->n->p = i->p;
   // delete i;
    }
}


void list::append(vehicleP veh)
{
  list *i = new list();

  i->v = veh;
  i->n = this;
```

5. **Run the application.**
   When Freeway is displayed, start the application and let it run. It should stop when the first vehicle reaches the end of the lane.

6. **Check for leaks during execution.**

   a. **Type** `showleaks` **in the command pane.**
   No leaks are detected up to this point.

```
(debugger) showleaks
Checking for memory leaks...
Leak Summary:
        actual leaks:         0  total size:       0 bytes
        possible leaks:       0  total size:       0 bytes
        blocks in use:     2378  total size:  278247 bytes
```

   b. **Continue the application by clicking on** `cont`**.**
   Allow the application to continue past both breakpoint by clicking on `cont` once.

c. Type `showleaks` **in the command pane.**

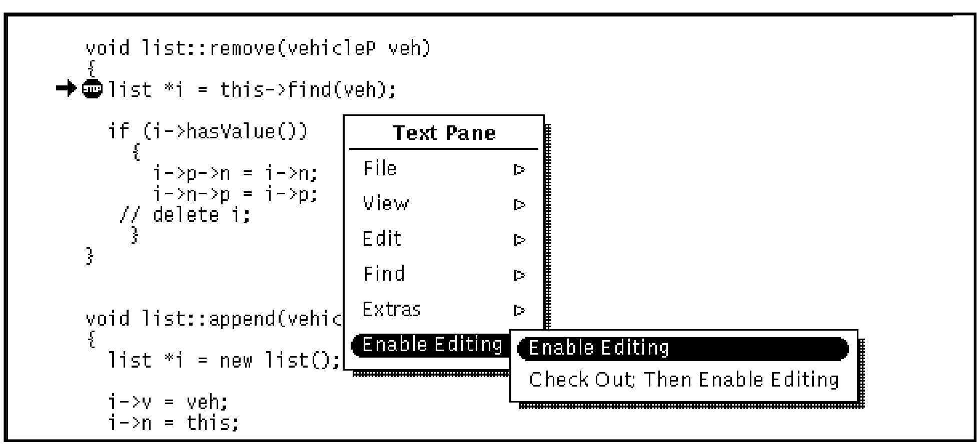
```
(debugger) cont
(debugger) showleaks
Checking for memory leaks...
Memory Leak (mel):
Found leaked block of size 12 bytes at address 0xa72e0
At time of allocation, the call stack was:
        [1] operator new() at 0x309b0
        [2] list::prepend() at line 62 in "vehicle_list.cc"
        [3] traffic_generate() at 0x1d22c
        [4] traffic_simulate() at 0x1cbb4
        [5] notify_itimer() at 0xef6d2424
        [6] ndis_default_prioritizer() at 0xef63e610
        [7] notify_client() at 0xef664168
        [8] ndis_default_scheduler() at 0xef6d29e4

Leak Summary:
        actual leaks:         1  total size:      12 bytes
        possible leaks:       0  total size:       0 bytes
        blocks in use:     2385  total size:  224119 bytes
```

The `showleaks` command shows any new leaks since the last `showleaks` command was invoked. One leak is occurring. Now see if the line that is commented out is the causing the leak.

7. Fix the error.

a. Edit the code

Put the cursor in the source display and choose Enable Editing from the Enable Editing submenu in the Text Pane menu (hold down the Menu button to display this menu).



b. Delete the comment and save the change by selecting Save Changes from the Disable Editing submenu item in the Text Pane menu.
The Enable Editing menu item toggles to Disable Editing.

c. Clear all breakpoints.
Choose Clear All Breakpoints from the Breakpoint menu.

d. Choose Fix from the Execution menu.
When you invoke Fix, the modified file `vehile_list.cc` is compiled and a shared object file is created. The pre-modified file and the modified file are compared. The new object file is then linked to the running process via the runtime linker. The modified file is displayed in the source pane and now you can check to see if the memory leakage has been corrected.

8. Run the application
Click on the Start button in the application's window to begin the program. Let the first car complete one loop and then click on the Stop button.

9. Quit the application and check for errors.
Quit the application from the Freeway window menu and look in the Debugger's command pane for a summary of errors:

```
Checking for memory leaks...
Leak Summary:
        actual leaks:         0  total size:       0 bytes
        possible leaks:       0  total size:       0 bytes
        blocks in use:     1520  total size:  149925 bytes
```

The message shows that no leaks were found.

For a thorough discussion of runtime checking, memory access errors, and memory leaks, see the section on runtime checking in *Debugging a Program*.

If you want to rerun the runtime checking exercise again, replace the modified `vehicle_list.cc` with it's original "flawed" version. A copy of the flawed file, named `vehicle_list.cc.orig` is included in the sample program directory. You will have to rebuild the application using the original version of the file so that program will again have memory leaks.

2.4.9 *Collecting Performance Data*

Solaris

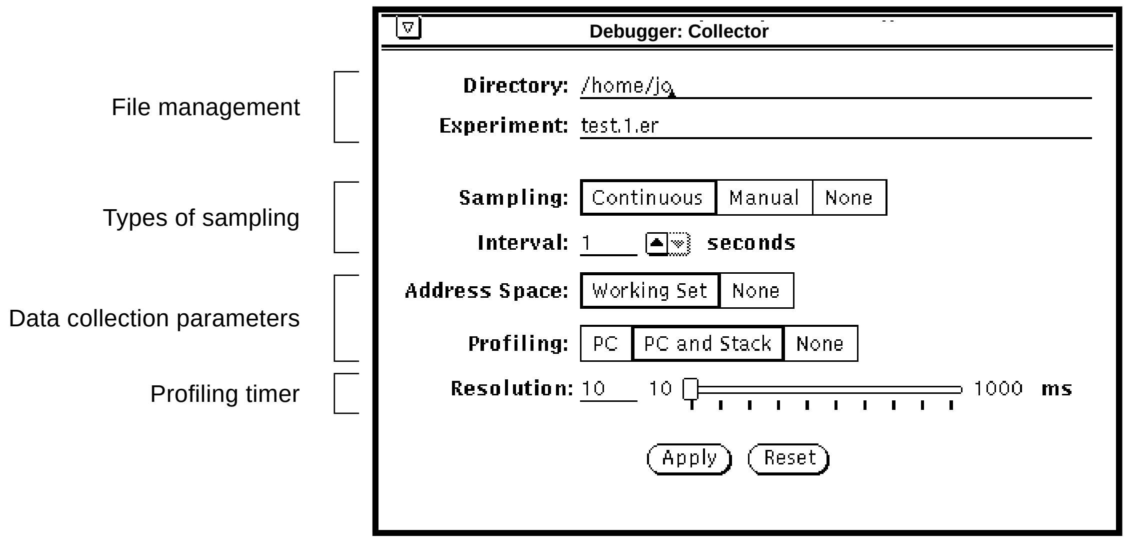
The Debugger's Collector and the Analyzer tool are available only on platforms running Solaris 2.x.

This part of the exercise requires you to collect data from the Debugger. You collect data by running an *experiment*. You specify an experiment by marking the sections of the program that the Debugger will monitor during the experiment. You can do that by setting breakpoints at the start and end of the test section or by specifying the program functions to be tested. The Debugger saves the data it collects in a file called an *experiment record*.

You will then load the performance data into the Analyzer for use in "Exercise 5 — Improving Freeway Performance" on page 59.

1. **Start the Debugger (with RTC disabled) and load the Freeway program.**
   Profiling data collection is disabled when RTC is enabled, so you need to disable RTC by choosing No Checking from the Error Checking submenu. You can also start a new Debugger from the Manager or by typing `debugger &` without the `-C` option at a shell prompt.

2. **Collect performance data from the Freeway program.**
   Open the Collector window by clicking on Collector from the Execution menu in the main window.



The Collector provides a default experiment name, `test.1.er`. You can change the prefix of the file name to anything you want, but keep the `.1.er` suffix. If you decide to create another experiment, the Collector will automatically increment the file name by one, for example `test.2.er`. The naming scheme helps you to keep track of the experiments.

When an experiment is created, a hidden directory is also created. The hidden directory name is preceded by a dot (.); for example, if the experiment file is called `test.1.er`, the hidden directory is called `.test.1.er`. Files in this directory contain information on segments, modules, lines, functions, sample, and more. For more information, see *Performance Tuning an Application*.

3. **Set the sampling properties in the Collector.**
   Samples are sets of data collected over specific periods of time while an application is running. The samples you take provide you with performance information, such as resource consumption and system calls.

   File management — Has text fields where you can change the directory name and the file name where the data will be stored. Specify the directory where you want to store the experiment file.

Types of sampling — Has settings that specify the period of time to collect data. Select Continuous and leave the Interval setting at one second (the default). This setting allows you to take samples as you observe the running Freeway application.

Data collection parameters — Has settings for specifying the types of data to collect. Select Working Set to get information about page usage. Select PC and Stack to generate performance data for cumulative histograms.

Profiling timer — Determines the number of profiling time packets that are obtained in the sample. Leave this setting at its default value of 10 milliseconds.

For detailed information on collecting samples, see *Performance Tuning an Application*.

4. **Apply all your parameters by clicking on Apply.**

To free up workspace, close the Collector window to an icon. As the collection process continues, the hands on the clock rotate continuously.

5. **Run Freeway.**
   Click on the run command button in the Debugger window. When you run the Freeway application, the collection process begins.

   You can also select Run from the Execution menu or type `run` at the Debugger prompt.

6. **Start the application.**
   Click on the Start button in the Freeway window and allow the first car entering the freeway to complete one loop. Click on Stop when the car exits the freeway.

7. **Stop data collection.**
   Quit the application to stop the collection process. You can quit the Collector, too. If you don't want to take a second sampling, you can quit the Debugger as well.

   To take a second collection, bring up the Collector, set the properties, and run the application again. You might try changing some of the Freeway parameters, such as reducing the time between updates.

In the next exercise, you use the Analyzer to examine the data you collected in this exercise. You can either go on to the next exercise or stop here.

You'll be loading the `test.1.er` file you just created into the Analyzer. You might want to go to the directory you designated for the experiment file and check to see that it exists. You should also look in `.test.1.er` to see if the following files exist:

functions	lines	overview	segments	working_set
journal	modules	profile	strings	

If `test.1.er` does not exist, you should repeat steps 1 through 7.

2.5 Exercise 5 — Improving Freeway Performance

Solaris

The Analyzer is available only on platforms running Solaris 2.x.

You've run the Freeway program, changed its settings, and examined function calls, but how do you know if it's running at optimum efficiency? Is there some portion of the program that is slowing the program down? Is too much time being spent initializing the program? By analyzing the performance of the program, you can thoroughly search for areas that are in need of improvement.

You could use the UNIX `prof` and `gprof` performance profiling tools, but those tools will only yield user CPU information. By loading a program's performance data into the Analyzer, you can get information about I/O time, system time, text and data page fault times, program sizes, execution statistics, and more in addition to user CPU information.

Another benefit of using the Analyzer is that it allows you to look at data accrued over a specific period of time during program execution. For example, if you wanted to see how much time is spent in the execution of your instructions without getting information on initialization time or time spent on system calls, you can select the User Time data type and see specific user time information.

Goals — In this exercise, you learn how to

- View performance data in various displays
- Examine specific types of data

2.5.1 Examining Process Time Data

The Analyzer uses four displays to present the collected performance data: Overview, Histogram, Address Space, and Statistics.

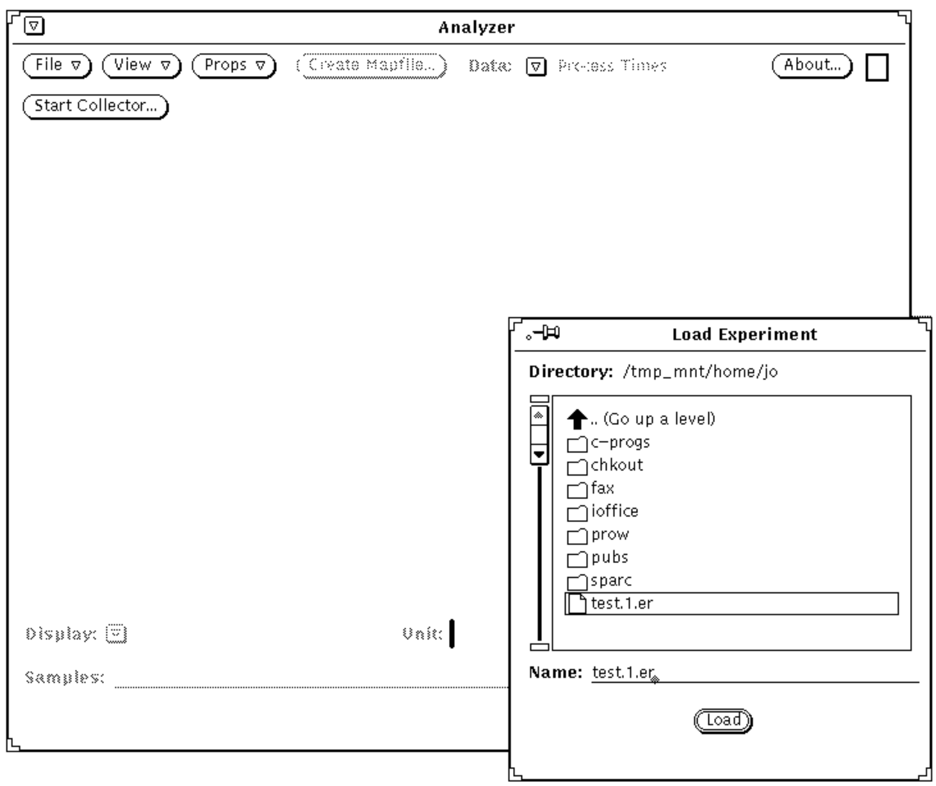
- **Overview —** Gives an overview of the performance behavior of the application.
- **Histogram and Cumulative Histogram —** Provide information about resources used by functions, modules, and segments of the application.
- **Address Space —** Shows what areas in the address space that the application occupies.
- **Statistics —** Provides statistical information about the application attributes that are not displayed in any of the other displays.

For a detailed discussion of all Analyzer displays, see *Performance Tuning an Application*.

1. **Start the Analyzer by clicking on its icon in the Manager palette.**
   You can also start the Analyzer by typing at the shell prompt:

   ```
   analyzer &
   ```

   The main window of the Analyzer and its Load Experiment window open simultaneously. The main window remains inactive (an incomplete main window is displayed) until you load a experiment.

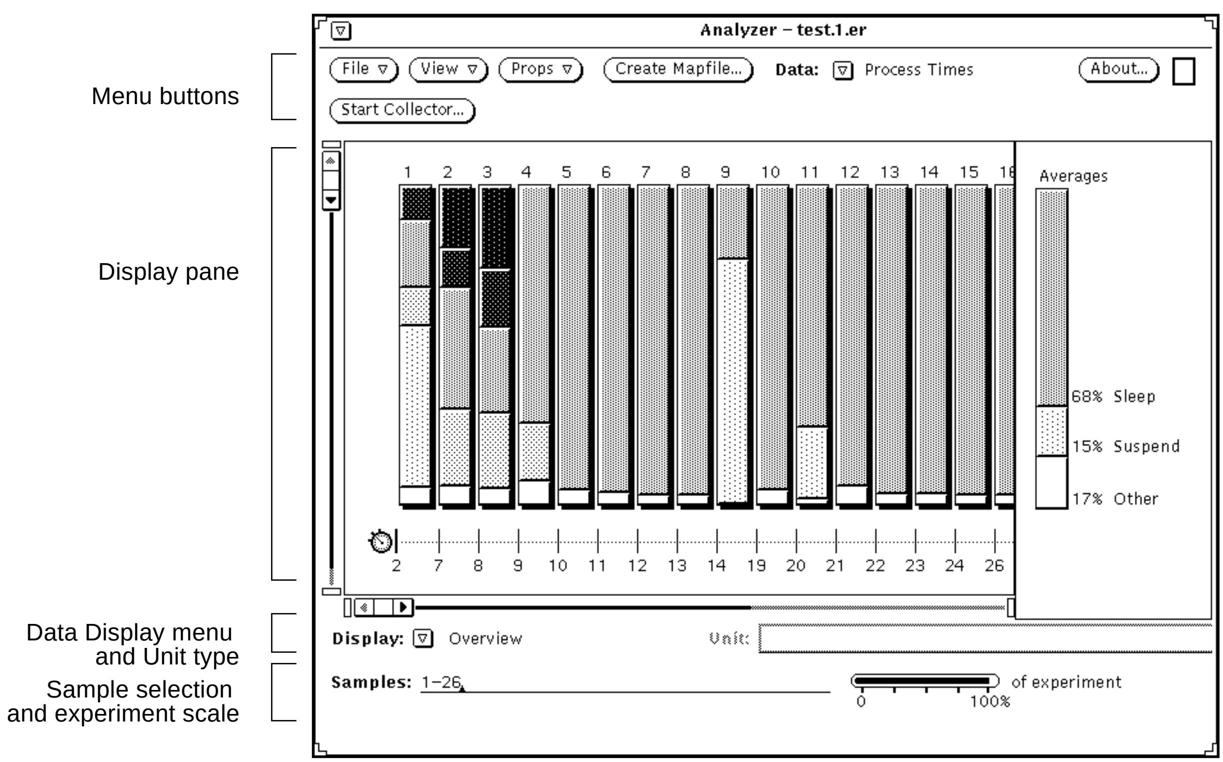


2. **Load the experiment file you created in the last exercise.**
   Double-click on the experiment file name in the Load Experiment window. If you do not have an experiment file, go to Section 2.4.9, "Collecting Performance Data," on page 2-55 in "Exercise 4 — Debugging Freeway" and follow the steps to create one.

   You could also load the file by typing in the path to your `test.1.er` file in the Load Experiment window and clicking on the Load button.

   Alternatively, if you have the File Manager running, you can drag the experiment file icon onto the Analyzer's Drop Target (in the main window) to load it.

3. **View Process Times information in the Overview display.**
   This is the default display that appears when the experiment file has been loaded.



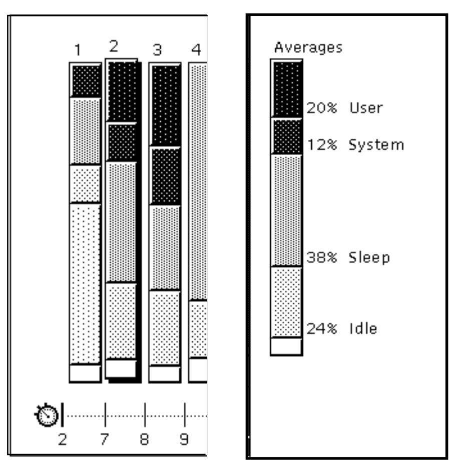
Note – Your samples will differ somewhat from the samples shown here because of differences between the state of your machine and the state of the machine that ran the experiment from which the samples shown here were taken. The number of samples taken may also differ.

The Overview display shows you

- Number of samples that were taken during the collection process
- Breakdown of the process activity for each sample during the collection process
- Breakdown of the process activity averaged over selected samples
- Percentage of the entire experiment that you're viewing

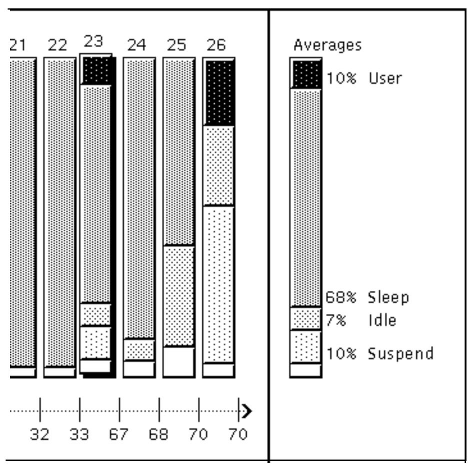
From the display, you can see some system and user time activity in the first three samples indicating initialization time. Samples 4–22 show mostly idle time; the application is running but hasn't been started yet. Sample 23, in this example, shows some user time again (this is where you actually started the application). The last sample shows the process being shut down.

4. **Examine sample number 2.**
   Find sample 2 (number designations are located above each bar) and select it by clicking on it.



The Averages pane shows you how much time was spent in each activity for the selected sample.

5. **Examine sample 23.**
   Scroll to the end of the experiment and select sample 23.



Look at the averages for sample 23 and compare them to the averages for sample 2.

Since the application had been initialized, no system time is shown in sample 23.

You can use the horizontal scroll bar to move back and forth across the samples and you can also select samples by typing the sample numbers in the Samples text field and pressing Return. When you're done viewing samples, choose Select All from the View menu to restore the view of the entire set.

6. **Look at a sample in proportion to its duration during data collection.**
   Choose Proportional from the Column Widths item in the View menu. You can see that samples collected during the initialization, the start, and the termination of the application are significantly wider than the rest of the samples. The proportions are also indicated by the time line. With this view of the samples, you can do a visual comparison of the duration of samples.

7. **Return to the fixed-width view.**
   Choose Fixed from the Column Widths item in the View menu.

2.5.2 Examining User Time Data

While the Overview display allows you to see the application's overall behavior, the Histogram display allows you to see the applications behavior in terms of functions, modules, and segments. You can examine where the application is spending most of its user time. In this part of the exercise, you'll see histogram displays for functions, modules, and segments.

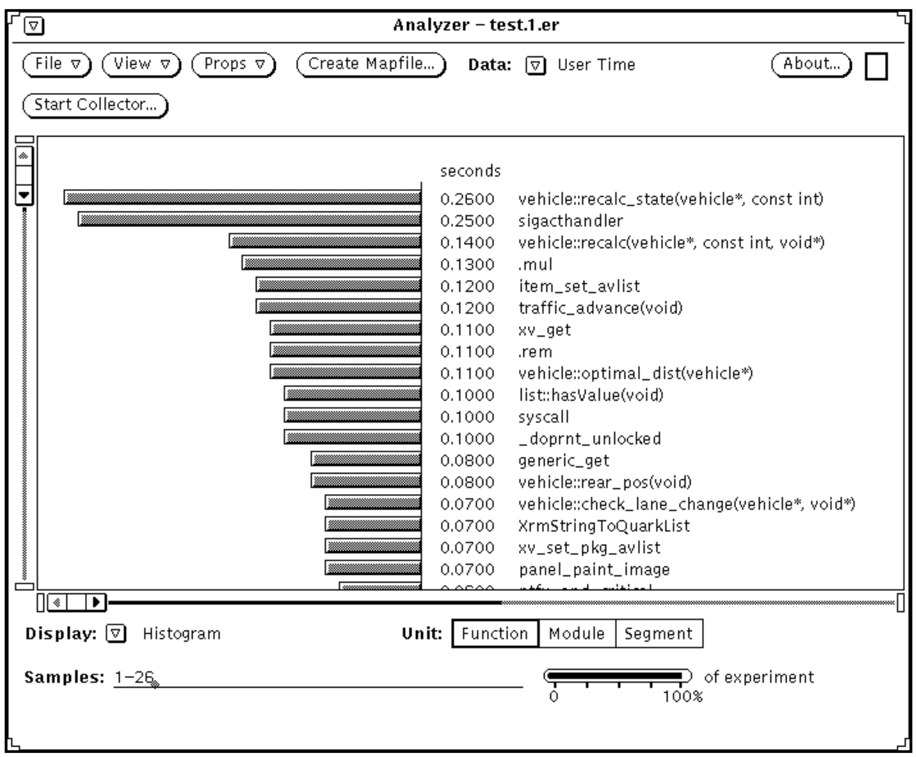
Function histogram — displays the amount of time the application spent executing in each function.

Module histogram — displays the amount of time the application spent executing in each *module* (group of functions from a single source file).

Segment histogram — displays the amount of time spent executing in each segment of the application. The segments of the program are its text segment and all of the dynamically-linked shared libraries.

See *Performance Tuning an Application* for a detailed discussion of each type of histogram.

1. **Look at the User Time display by selecting User Time from the Data menu.**



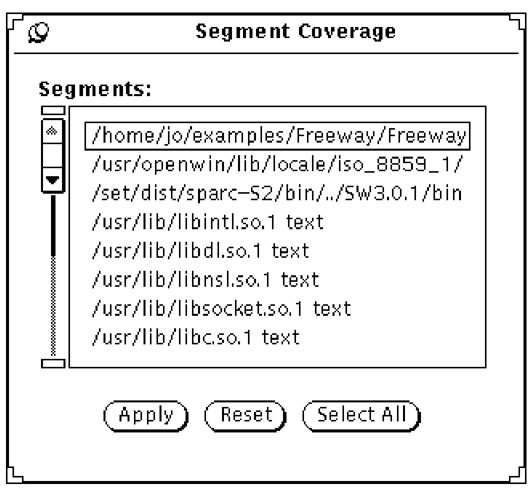
Data is displayed in a histogram instead of a bar graph. You can see in which routines the application is spending most of its time:

- First column indicates graphically the amount of time spent executing the corresponding functions.
- Second column gives the time spent executing the corresponding functions.
- Third column identifies the functions.

You can also see what portion of the total experiment you're looking at from the experiment scale at the bottom of the display. In this example, approximately 50% of this example is shown.

The default display includes library files. For this part of the exercise, you should filter out the library files in the display to view only the text segment of the application. Later on you'll add the libraries back to the display.

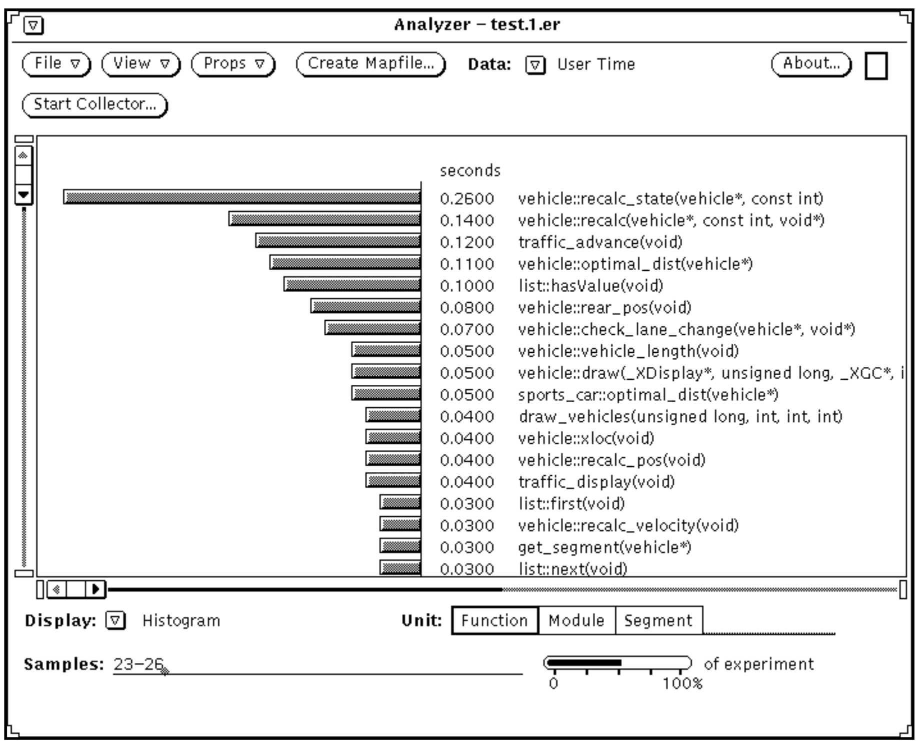
2. **Choose Segment Coverage from the File menu.**
3. **Deselect all but the first segment in the Segments pane (which should be one of the Freeway paths that you are using):**
    - `/your_dir/SW3.0.1/examples/Freeway/Freeway text`
    - `/opt/SUNWpro/SW3.0.1/examples/Freeway/Freeway text`



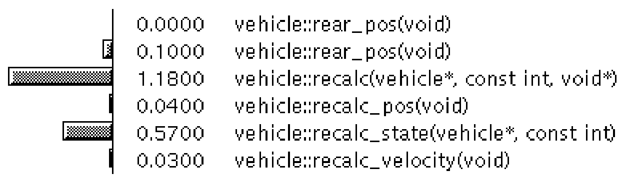
4. **Click on Apply.**
   All the library files are removed from the display.

   Now let's concentrate on the samples at the end of the experiment, the portion that covers the actual running time of the application.

5. **Select samples 23 through 26 by typing 23-26 in the Samples text field and pressing Return.**
   The second bar in the histogram display now shows the execution time for `vehicle::recalc(vehicle*, const int, void*)`:



6. **Display the information by function name instead of value.**
   Choose Name from the Sort by item in the View menu. The display is rearranged to show the function names in alphabetical order. Try scrolling through the display to see which functions were executed.

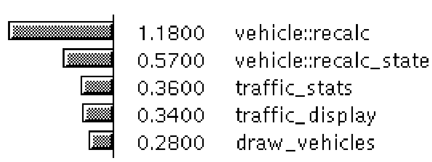


If you want to look at a specific function, you might be able to locate it quicker in this arrangement. This view also helps you to compare the same functions in two experiments.

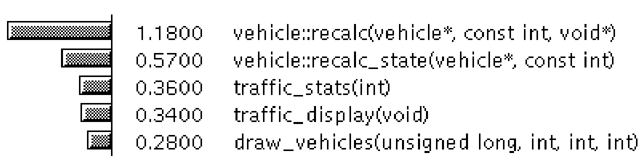
You can also locate a specific function with the Find command in the View menu (see step 10).

7. **Return the view to the value display.**
   Choose Value from the Sort by item in the View menu.

8. **Show the shortened function names.**
   Choose Short from the Names item in the View menu. The default display shows the function names including their arguments. The short name display shows just the function names:



Names Short



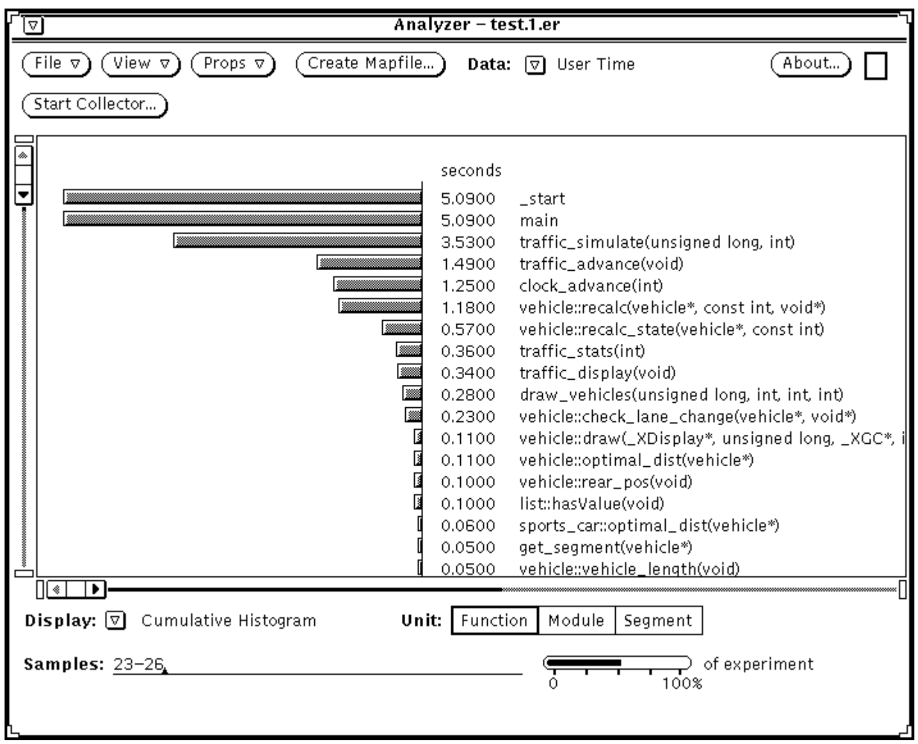
Names Long

9. **Revert to the long name view by choosing Long from the Names submenu.**

10. **Search for the function** `vehicle::recalc_pos`**.**
Choose Find from the View menu and type in the function name. Click on Find Forward to display that function.

You can also search backwards through the display for a particular function. Both Find Forward and Find Backward have a wraparound feature, so that when it reaches the beginning or the end of the histogram, it continues the search.

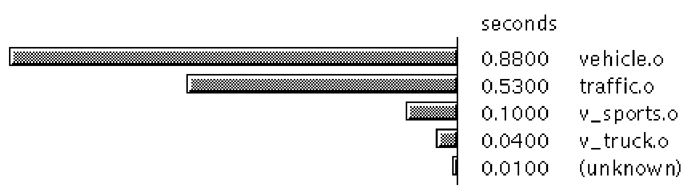
11. **Look at the Cumulative Histogram display.**
Scroll back to the top of the Histogram display and then choose Cumulative Histogram from the Display menu at the bottom of the window.



Display menu

This display allows you to see the total time taken to execute a function and the children of that function for the given sample set. For instance, look at the histogram in step 5 for the execution time of `vehicle::recalc(vehicle*, const int, void*)` and compare it to its execution time in the cumulative histogram. The histogram shows that `vehicle::recalc` took .14 seconds to execute; however, the cumulative histogram shows you that it took over 1 second for `vehicle::recalc` and all of its subroutines to execute. The cumulative histogram points you to potential problem areas in the program.

12. **Change the Unit value from Function to Module.**
    You can see the amount of time that was spent executing each module in the application.



Related functions are grouped together into a single source file (called a *module*). Viewing modules can provide you with a summarized data display of application performance. The preceding figure shows that most of the application activity occurred in `vehicle.o`.

If the application was not compiled with the `-g` option, the Debugger might not be able to associate some functions with some modules. In such cases, a module is displayed as *unknown*.

13. **Change the Unit value from Module to Segment.**
    This display allows you to see the amount of time that was spent executing text segments in the application.

    a. **Choose Segment Coverage from the File menu.**
       To examine the performance for libraries, you must add the information back into the display. You need to bring up the Segment Coverage window and select the library files that you want included in the display.

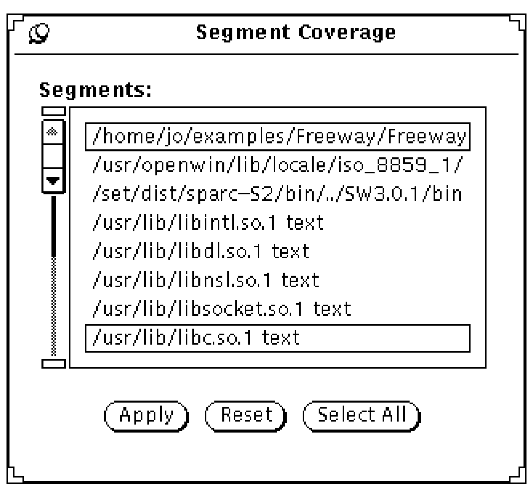
    b. **Click on Select All and then click on Apply.**
       The display shows the execution time for the main application, the C library, and other shared libraries.

14. **Look at the routines called by Freeway and the** `libc` **library.**

   a. **Switch Unit from Segment back to Function.**

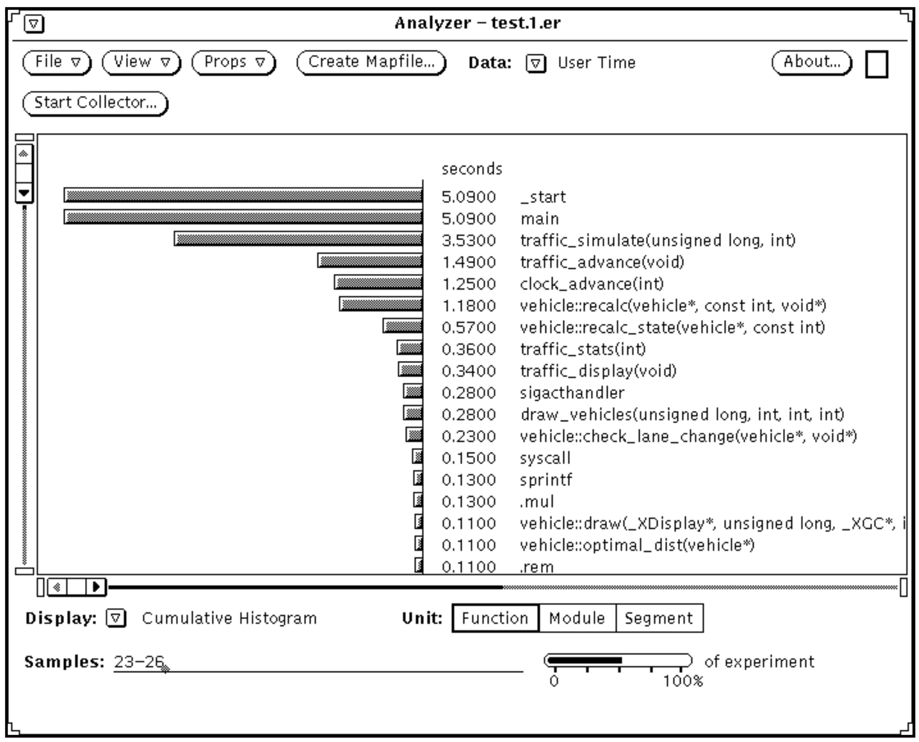
   b. **Choose Segment Coverage from the File menu.**

   c. **Deselect all the libraries except** `/usr/lib/libc.so.1 text` **and your Freeway path.**
   To deselect a library, click on the library path. To deselect multiple libraries, put the cursor over a library path, and drag the mouse over the desired paths.



d. Click on Apply and look at the display pane.

Compare this display with the Cumulative Histogram display in step 11. The display now includes library routines, another possible area of optimization. Look down the list of function names and you see `sigacthandler`, `syscall`, and a little farther down, `.mul` and `.rem`.



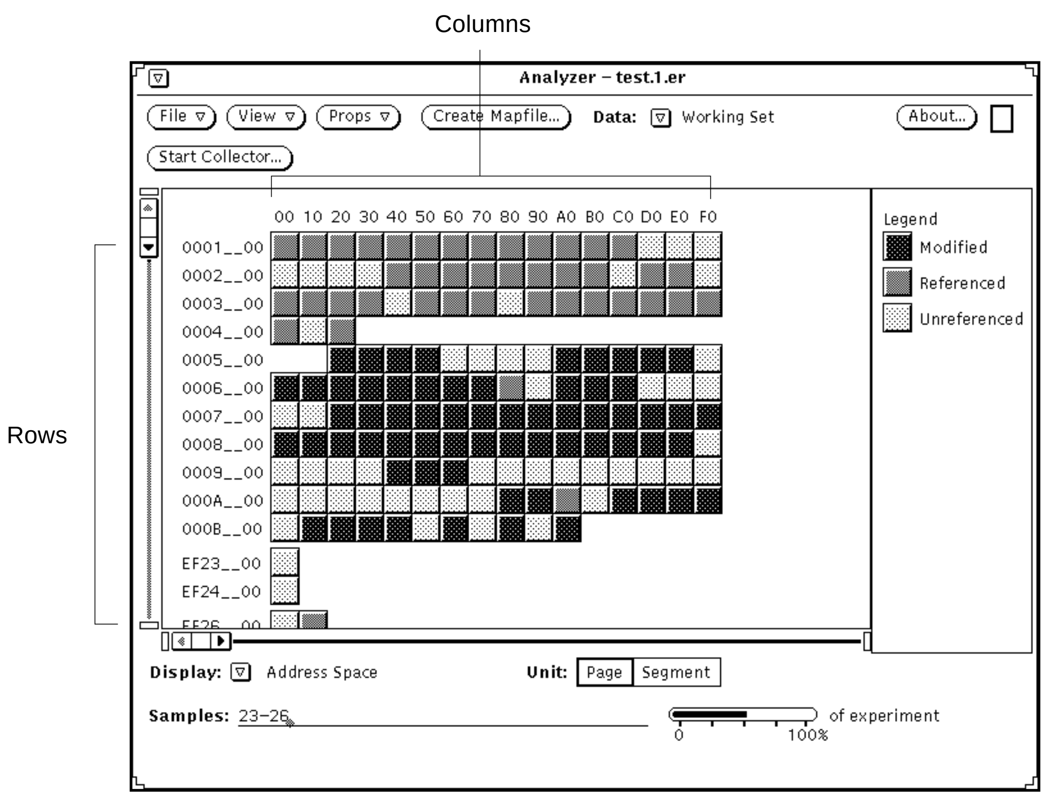
In this particular example, the display shows that little time was spent in handling signals or system calls. If large amounts of time were spent on library routines, you could spot them quickly here.

2.5.3 Examining Working Set Data

The Address Space display represents pages and segments of memory in the address space as individual squares and blocks. The pages and segments represent the application's memory usage. With Unit type set to Page, you can pinpoint areas of memory that were changed, read, or unused by the degree of shading in each square.

For information on address space, see *Performance Tuning an Application*.

1. **Change the Data type from User Time to Working Set.**
   The display changes to show address space. The legend to the right of the display denotes the memory usage categories.



Gaps (white space between the pages) represent regions of the address space not used by the program.

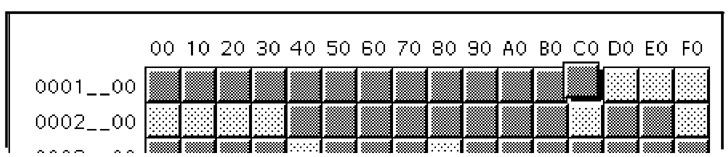
Modified pages are pages that were written while the application was running. Referenced pages are pages that were either read by the application or contain instructions executed by the application.

Unreferenced pages are pages that represent allocated but unused memory. These pages can also denote dead code or problems with memory allocation.

In general, it's better to have fewer referenced and modified pages. Having fewer modified pages is even better since modified pages are more costly than referenced pages.

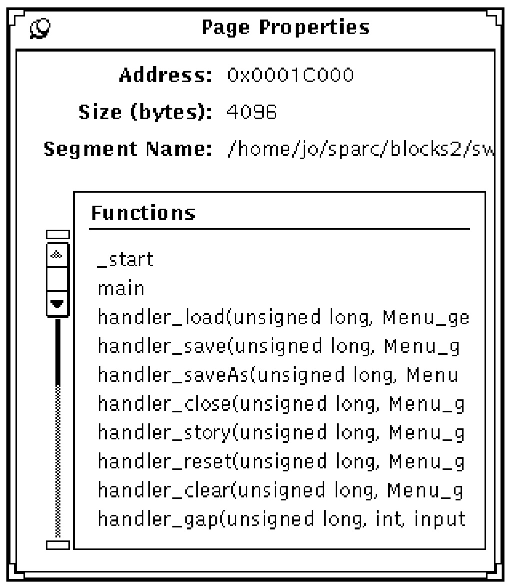
2. Examine a page of memory.

a. Go to the first row of pages and select the page in column C0.



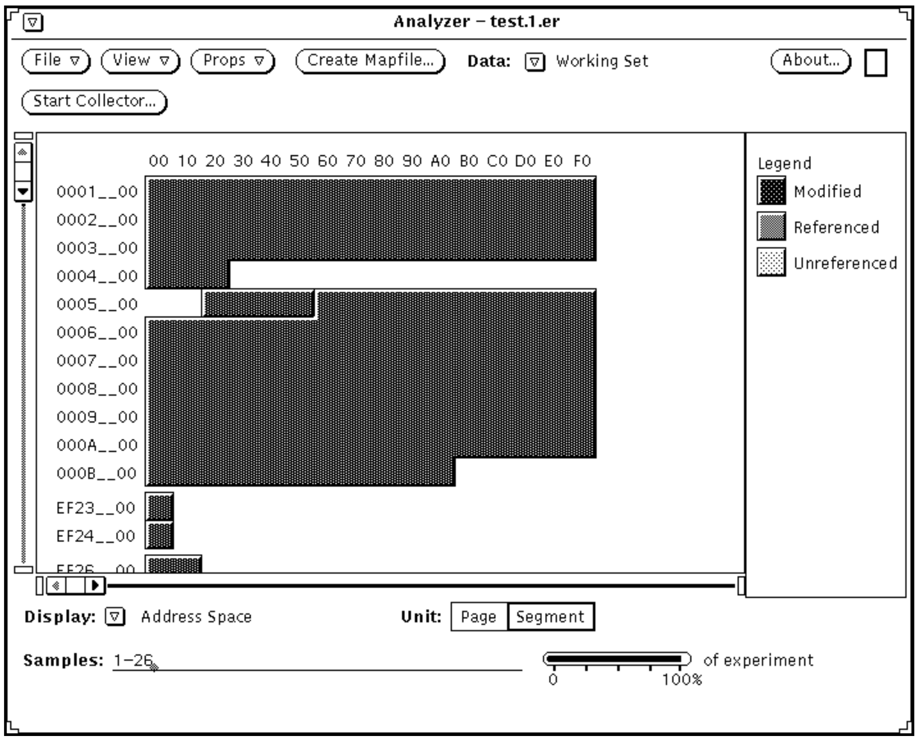
This is the page that starts at hex address 0X0001C000, which is obtained by inserting the column header value in the blanks in the row header value.

b. Click on the Props button to bring up the Page Properties window.

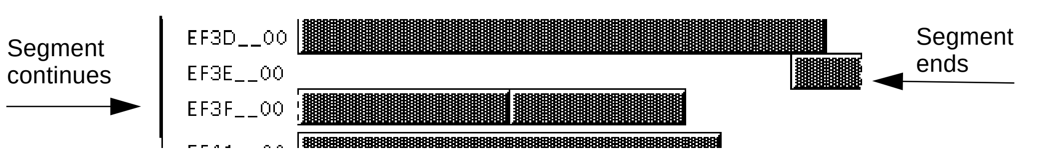


This window gives you the hexadecimal address of the page, the size of the page, the functions contained in the page, and the segment name.

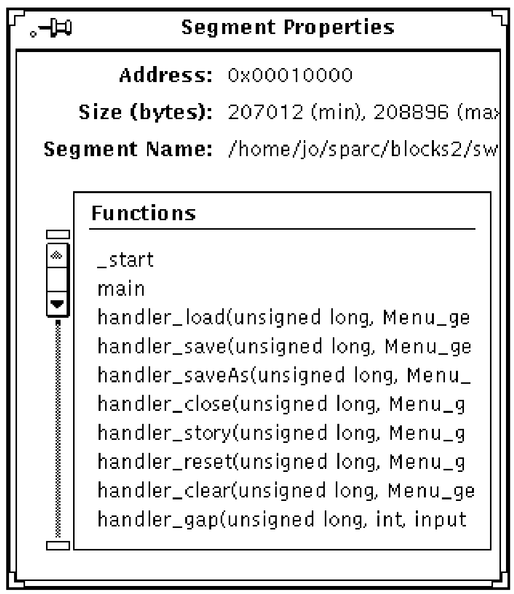
3. **Now examine a segment of memory.**
   Change Unit type to Segment. The individual pages disappear and are replaced by blocks. The legend in the right pane only applies if you select a segment block while holding the Select button down. The block shows the individual pages shaded just as if you were in page mode. This is helpful if you want to know which pages the selected segment is made up of. The segment display gives you a high-level view of the mapping of segments into the application's virtual address space.



Sometimes a segment starts in one row and continues into the next row. The continuation of a segment is denoted by a broken line at the end of one row and at the beginning of the next:



4. Look at the segment properties for the first block of memory.
Select the first block and click on the Props button.



You can find out what functions are contained in a particular segment by looking in the Segment Properties window. The first block in this experiment includes the `_start` and `main` functions.

The first segment of the application is a good place to look for areas of possible optimizations.

5. Click on the second segment and expand the Segment Properties window so you can read the full segment name.
Note that the second block is the data segment; there are no functions. Click on a few more segments and see what they contain.

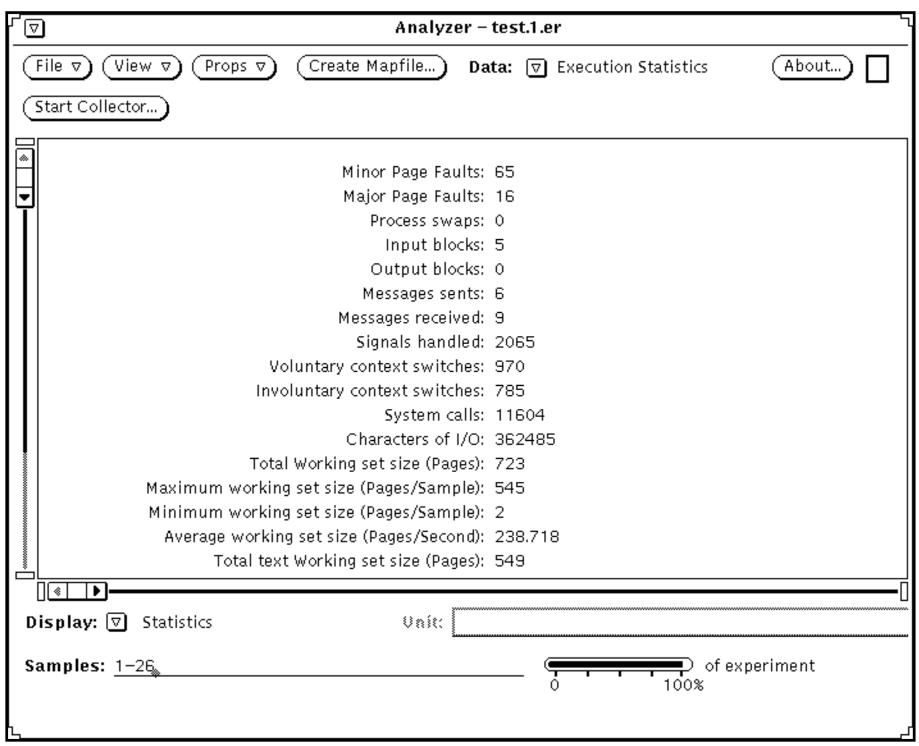
6. Click on a segment and then change the unit type back to Page.
Looking at segments first allows you to quickly locate a particular library by selecting each segment. You can then switch the display units back to pages to find out what routines in the library a page in that block contains.

Reminder: you can see the pages in a segment without changing the unit type. Select a segment and hold down the SELECT button. The pages in that segment remain visible until you release the button. The reverse also works; that is, by holding SELECT down with the cursor over a page, you can see the segment it is a part of.

2.5.4 Examining Execution Statistics

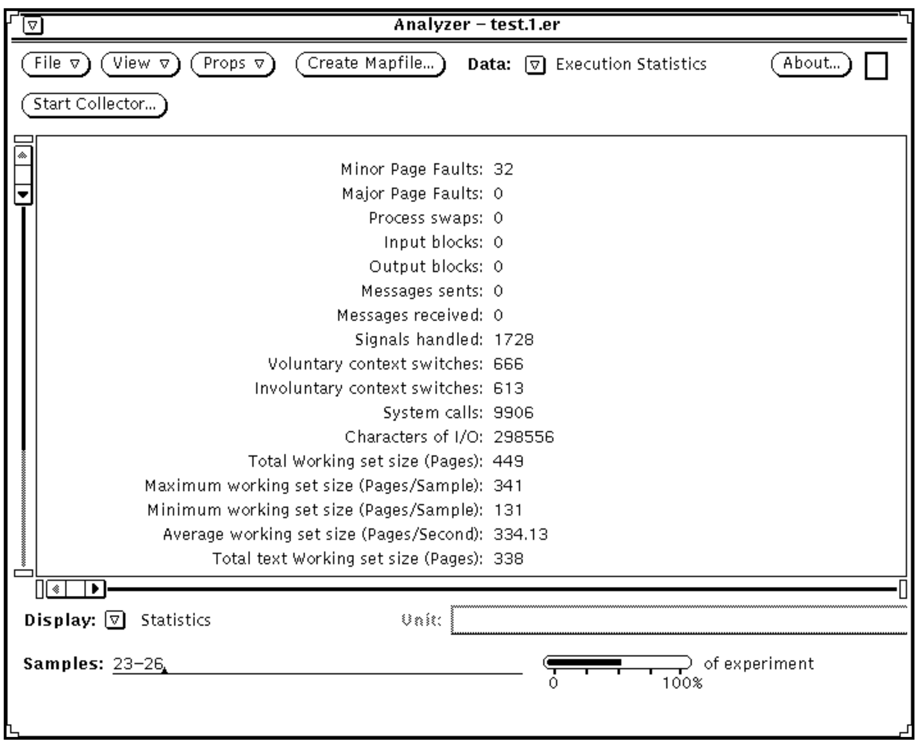
The Statistics display provides information on application attributes that is not shown in the other displays. You can obtain actual values for these attributes, such as exactly how large the working set size is, the number of signals handled, the number of system calls that were made, and more. You can use the values in this display to corroborate any estimates you might have made about particular attributes. You can also look at this display to see if you need more swap space or more memory.

1. **Look at the execution statistics for the entire experiment by changing the Data type to Execution Statistics.**
   The display changes, showing a list of activities and their numerical values.



2. **Compare the execution statistics of the entire experiment with those for samples 23 through 26.**
   You can select all the samples by typing the full range of samples (1-*n*) in the Samples text field and pressing Return or by choosing Select All from the View menu when you have the Overview display up (Data type is set to Process Time); then change the Data type to Execution Statistics.

   Then type a smaller range from the end of the experiment (for example, 23–26) in the Samples text field and press Return. The display shows the values for the application's attributes for just those samples.



   You can also select specific samples by clicking on the samples you want in the Overview display and changing the Data type to Execution Statistics.

   For more information on the Statistics display, see *Performance Tuning an Application*.

3. **Quit the Analyzer and delete the experiment file.**
   To delete the current experiment file and the associated hidden files, use the `er_rm` command in the directory containing the files (remember to quit the Analyzer first); for example:

   ```
   er_rm test.1.er
   ```

   To delete previously created experiment files (along with the associated hidden files), choose Delete from the File menu and double-click on the experiment file name from the Delete Experiment pop-up window.

Congratulations, you have completed the tutorial. You should now be able to open and operate the following tools: Manager, MakeTool, Debugger, and Analyzer. You have performed simple operations with each tool. Some tools, such as the Debugger and the Analyzer, are loaded with more features than it is practical to demonstrate in a tutorial. See the individual tool manuals for detailed information on how to use each tool.

Part 2 — Tools Overview

The Toolset and the Software Development Cycle 3

This chapter is organized into the following sections:

3.1 Programming Tools

As every programmer knows, the task of creating high-performance, bug-free software has become more difficult as applications grow larger and more complex. Innovations such as object-based design and object-oriented languages have helped somewhat to manage the demands of increasing complexity.

The tools form part of an overall development environment that includes the standard operating system programming utilities, the DeskSet™ productivity tools (including File Manager and Mail Tool), and the OpenWindows Developer's Guide for interactively building OpenWindows graphical interfaces.

These tools simplify the tasks programmers perform most often: coding, compiling, debugging, and performance tuning programs. Each tool is designed to fill a specific need and at the same time cooperate with the other tools in the set. The result is a flexible, integrated toolset that solves the most important problems faced by the individual programmer.

The remainder of this chapter explains which tools are useful at each stage of the software development process.

3.2 Conceptualizing

The early steps in any software design project are conceptual. The first step is to clearly define the problems that the software is meant to solve; the second is to propose solutions to these problems. The proposed solutions give rise to a specification, set of core algorithms, and an overall design approach.

For procedural programming, the result of the conceptualization stage is a detailed set of tasks that the program must perform, divided into functions that will implement the program's algorithms. For object-based programming, the result is a hierarchy of classes, a listing of the responsibilities of each class and the ways objects of each class will collaborate with other objects to fulfill their responsibilities.

This early stage of software design is arguably the most critical to the long-term success of a project. Regrettably, it is also the stage that is least amenable to computer-aided assistance, relying as it does on a clear assessment of the problem by the designers and an analysis of trade-offs between desired features and available resources.

Because SPARCworks and ProWorks concentrate on helping the individual programmer to write efficient code quickly, they leave assistance in such tasks as high-level program design, large team development efforts, and revision control to other computer-aided software engineering (CASE) tools.

3.3 Coding

Because each developer has a favorite editor, an editor is not included in the toolset. If you choose to use the Manager, you can place the editor of your choice in iconic form in the Manager window so that you can easily start it the way you do other tools. In some cases, you can associate an instance of the editor with a particular programming session.

After the compile-debug loop has begun, debugging tools provide editing windows so that you can make changes to source code quickly within the tool, recompile, and continue debugging.

3.4 Building Executables and Libraries

Most everyday compilation and linking on Sun systems is done with `make`, the utility that ensures that programs are built from the newest sources according to lists of rules contained in makefiles. While the value of `make` in maintaining large projects is widely recognized, writing and maintaining a makefile has always been an annoying penalty to pay for its use.

MakeTool simplifies the use of `make` and makefile maintenance. It expands makefile rules and macros so that they can be easily understood. In addition, MakeTool automatically creates a menu of high-level targets whenever it loads a makefile. You can use the menu to build the targets. MakeTool updates the menu automatically whenever you enter options and arguments to the `make` command.

Analyzing tools (Debugger, SourceBrowser, and Analyzer) require that programs be compiled with special flags before the tools can be used to their full capabilities. These flags create special databases or symbol tables for the tools, and only compatible compilers can respond properly to these flags. The following compilers can be used with the tools:

For ProWorks:

- ProCompiler C / ANSI C
- ProCompiler C++
- ProCompiler FORTRAN

For SPARCworks:

- SPARCompiler C / ANSI C
- SPARCompiler C++
- SPARCompiler FORTRAN
- SPARCompiler Pascal

Other Sun compilers, such as Sun Ada, provide their own programming environments and can only interact in a limited way with SPARCworks or ProWorks analysis tools.

3.5 Debugging

After a program successfully compiles, the search for errors (bugs) begins. Two tools assist in the debugging effort:

- The Debugger, which examines the values of variables and address locations while you step through the program.
- The SourceBrowser, which enables you to find any or all instances of a specific function or variable in source code. SourceBrowser is especially useful for analyzing unfamiliar code that you have to maintain.

These two tools, one a dynamic analyzer and the other a static analyzer, form the heart of the toolset. Used together, they help you find and remove bugs quickly. The capabilities of these two tools are extensive, making their features difficult to summarize.

3.6 Merging Source Code

Versions of source files have a way of proliferating when more than one programmer is working on a project. In the past, programmers used the `diff`(1) operating system utility and their favorite text editors to merge different versions of the same file. The development of sophisticated window interfaces has made the `diff` command-line interface obsolete.

FileMerge provides a convenient way to merge source files or entire directories of source files quickly. You can load two files into FileMerge and merge them, or you can specify a third file, the *ancestor* of the two files (which are called its *descendants*). When you specify an ancestor file, FileMerge marks lines in the descendants that are different from the ancestor and automatically produces a merged file based on all three files.

3.7 Testing and Performance Tuning

After you have successfully compiled a program and eliminated its major bugs, you will want to evaluate its performance. The Analyzer measures and displays a program's performance profile, suggesting ways to improve performance. The Analyzer effectively supersedes operating system utilities such as `prof`(1) and `gprof`(1), providing capabilities not offered by any existing standard utility.

The Debugger collects performance data for the Analyzer and places the data in a file. The Analyzer then examines the collected data and presents its analysis in a variety of graphical and text displays. Because data collection relies on operating system calls, not on Debugger symbol tables, any program can be evaluated with the Analyzer — no special compiler options are required.

Note – For SPARCworks users, performance data can be collected only when the Debugger is running under SunOS™ 5.2 or later.

3.8 Maintaining Software

For every programmer who initially codes a program, several others inherit the task of maintaining it. Programmers who inherit code find SourceBrowser an indispensable tool for grasping the structure of an unfamiliar program. SourceBrowser's ClassGrapher and CallGrapher provide detailed overviews of a program's organization by graphing its class inheritance and function call trees.

Inherited bugs are difficult to fix, not only because inherited code is less familiar than code that you authored but because inherited bugs are recognized late in the design cycle and are usually subtler than bugs recognized earlier. The combination of SourceBrowser and Debugger form a powerful team for tracking down elusive bugs. For example:

- With SourceBrowser, you can browse source code and quickly find every instance of a particular variable, function, or object. The tool contains a ClassBrowser that performs the same functions for class attributes in C++ programs. If you find a questionable line of source code, you can move to the Debugger, set a breakpoint at that line, and examine the values taken on by a variable during execution.

- After you stop at a questionable function in the Debugger, you can use SourceBrowser to find how the function is defined. You can also find all the other places where it is used.

3.9 System Requirements

To run ProWorks/SPARCworks effectively, your system should have at least 16 Mbytes of memory and 40 Mbytes of free disk space.

3.10 Summary

This introductory chapter closes with a map that summarizes how the tools are typically used in the everyday activities of software developers. The figure on the following page has been adapted from *Managing the Toolset*, the manual that describes the Manager. The Manager itself is not shown in the figure — it controls the other tools and is not directly involved with writing, debugging, or testing code.

The figure illustrates how each developer uses an editor to write high-level code, compile it with the MakeTool, and then debug and test it with the Debugger, the SourceBrowser, and the Analyzer. FileMerge is used to merge an individual's source code with that of other members of the development group or with alternate versions of source code.

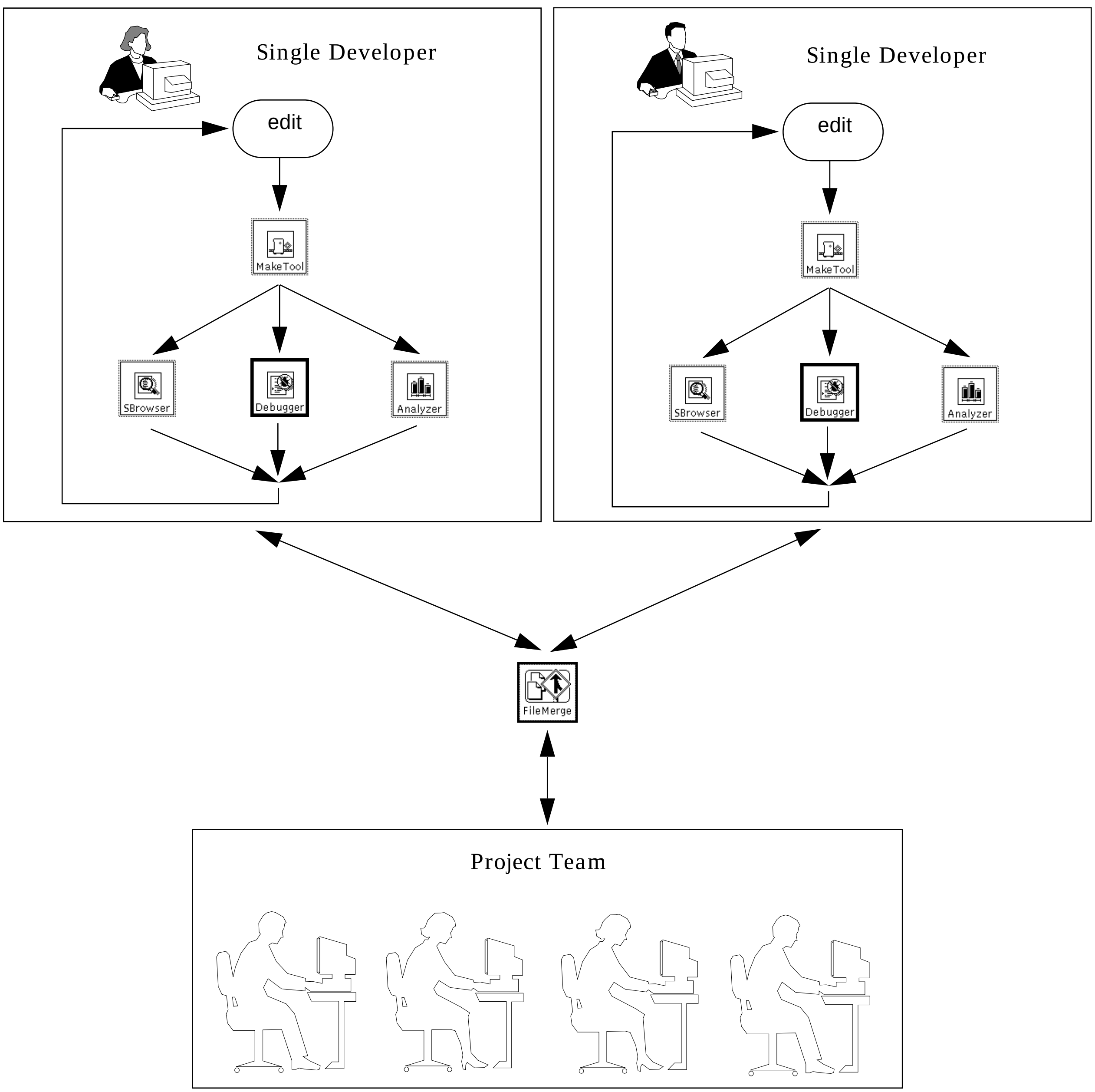


Figure 3-1 The Software Development Process

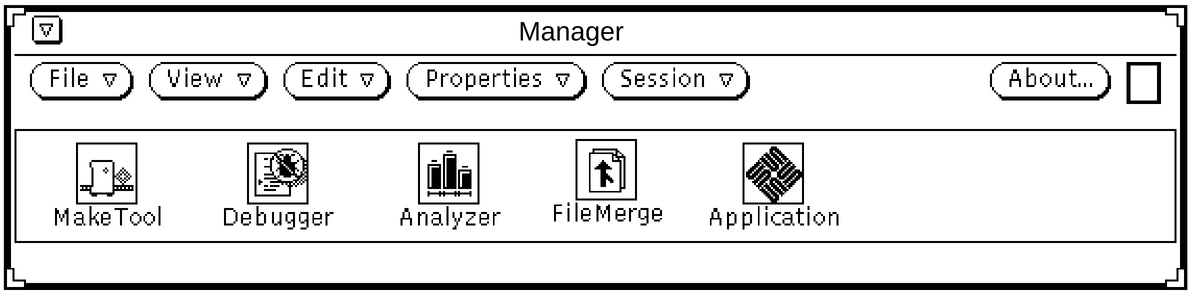
The Toolset 4

This chapter gives a brief overview of each tool in the toolset. For detailed information on the tools, you should refer to the tool-specific manuals provided in the documentation set. To get hands-on experience with the tools, go to the tutorial exercises in Chapter 2, "Tutorial Exercises."

This chapter is organized into the following sections:

4.1 Manager

The Manager coordinates the tools and controls the programming environment.The following figure shows the Manager window (also called the Manager palette) with the tool icons displayed:



Each tool can be started by double-clicking on its icon or by dragging the icon onto the screen workspace. A tool that has been started from the Manager is associated with that instance of the manager, and can be controlled as part of a management *session*.

Session control applies to any custom tools you have placed on the palette yourself, provided the tools understand the protocols of the ToolTalk software. This feature helps eliminate the confusion that results from having several instances of a tool active on screen at the same time and greatly reduces on-screen clutter.

You can also set properties that are common to all the session tools: the working directory in which new tools will be started and the start-up environment variables for the tools.

4.2 MakeTool

MakeTool is an OpenWindows interface to `make`(1), the SunOS utility that oversees program compilation and ensures that programs are compiled from the newest sources. MakeTool provides three major advantages over issuing `make` commands from a shell command line:

- Stores high-level makefile target names in a menu for easy access
- Provides visual feedback about the progress of a build even when closed into an icon
- Contains a browser that helps you interpret and debug a makefile by expanding its macros and rules, even when they are default rules or have been included from other makefiles; however, MakeTool does not generate a makefile for you

4.2.1 MakeTool Window

The MakeTool base window opens when MakeTool starts. MakeTool loads a makefile automatically, using the same search algorithm as `make`. If MakeTool finds a makefile, it loads and displays the file in the MakeTool window. You can, of course, load a different makefile after MakeTool starts.

Building the Target Menu

As MakeTool loads a makefile, it searches it for the first twenty high-level `make` targets. These targets are placed in a menu that you access by clicking SELECT on the Target menu button. You can build any of the targets by selecting it from the menu and then clicking on the Start Make button. The item you select replaces the string in the text entry field next to the Start Make button. The message displayed at the bottom of the window tells you whether the make succeeded or failed.

4.2.2 Makefile Browser

MakeTool contains a browser to help you interpret makefiles by expanding rules and macros. Note that the Makefile Browser shows rules and macros but does not provide the means to edit them.

Seeing the source of a statement is useful when the entire rule or macro is too long to be shown on a single line; the display of source wraps the line so that all of it is visible. If necessary, MakeTool formats the display for clarity by inserting spaces around operators.

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

4.2.3 Starting MakeTool from the Debugger

You can initiate program builds with `make` from the Debugger. You have the option of initiating the build directly or calling on MakeTool as the `make` interface. When the Debugger activates MakeTool, it passes MakeTool the following information to use during the build:

- Name of the makefile to use
- Name of the working directory
- Shell environment variables and their values

When MakeTool receives the information, it:

1. Starts a temporary subshell in which to run `make`.
2. Loads the specified makefile.
3. Changes to the makefile's working directory in the subshell.
4. Sets the necessary environment variables in the subshell.
5. Issues the `make` command in the subshell.

This scheme enables you to use MakeTool during debugging sessions when, for example, the load library path (`LD_LIBRARY_PATH` environment variable) has been set to a nonstandard directory.

4.3 Debugger

Through the Debugger, you can observe the behavior of a program while it is running. The Debugger gives you control of program execution and can simultaneously collect performance data for later use with the Analyzer. From within the Debugger you can identify a problem, edit source code, rebuild the program, and then continue inspecting its run-time behavior.

You are probably familiar with window-based source language debuggers such as `dbxtool`, the SunView™ interface to the `dbx`(1) command-line debugger. The Debugger is the successor to `dbxtool` and is based on `dbx`.

Besides being much easier to use than dbxtool, the Debugger differs from `it` in two major ways:

- It uses ToolTalk for inter-process communication with `dbx` and other tools in the toolset, while `dbxtool` uses pipes to communicate with `dbx`. ToolTalk provides more flexibility (drag-and-drop execution, for example) than pipes.
- Serves as the data-gathering front end for Analyzer, the performance analysis tool. You control the data gathering process with a popup Collector window in Debugger, then conduct a run-time "experiment" by running the program in Debugger. You then use Analyzer to identify performance bottlenecks in the collected data.

The discussion in the remainder of this section generally does not attempt to distinguish between the Debugger and dbx, but treats them as a single tool for dynamic analysis.

4.3.1 Debugger Features

The Debugger contains the following features to help you examine and debug source code:

- Displaying the Contents of Memory
- Displaying Source Code
- Single-Stepping through the Program
- Checking for Runtime Errors
- Examining Variables
- Setting Breakpoints
- Setting Tracepoints
- Setting Post-Break Modifiers
- Managing Events
- Navigating the Call Stack
- Inspecting the Call Stack
- Handling Overloaded Names
- Fixing Source Code and Continuing Program Execution
- Calling Functions
- Analyzing Live Processes
- Handling Signals
- Inspecting Threads
- Analyzing Dynamically Linked Libraries
- Analyzing Multiple Languages
- Reading Symbol-Table Information on Demand
- Analyzing Optimized Code
- Saving, Restoring, and Replaying Sequences of Commands
- Stepping out of Functions with a Single Command
- Monitoring the Values of Expressions and Variables
- Assigning Values to Variables
- Evaluating Arrays
- Checking Out Source from SCCS
- Displaying Command History
- Using Korn Shell Syntax

For a complete discussion of each of these features and how to use them, refer to *Debugging a Program*. For a brief description of these features, see Appendix A, "More About the Debugger."

4.3.2 Customizing the Debugger

You can customize the Debugger by modifying its menus or command buttons:

- Custom Command Buttons

  Command buttons appear between the source pane and the command pane. The buttons provide instant access to the most commonly used commands.

  The Debugger `button` command and its modifiers (issued in the command pane) let you change any of these button commands — deleting or adding to them as you like. When you add a button to the panel, you associate with it a `dbx` command string of your choice — when you click on the button, the command is echoed in the command pane. The Debugger automatically adjusts the size of the panel to accommodate new custom buttons, adding rows as necessary. For detailed information on adding buttons, see *Debugging a Program*.

- Custom Menu

  The Custom menu, to which you can add your own items, is located to the right of the Props button. There is no functional difference between adding an item to the Custom menu with the `menu` command or adding it to the row of command buttons with the `button` command. However, if you have many custom commands, you might prefer the menu because it uses less screen space. Also, keep in mind that any button menu can be pinned during a session so that all its items are continuously on display, just as they are on the button panel.

  Custom menu items, like custom command buttons, are not automatically preserved from one Debugger session to the next. You must place the `menu` commands that create them in the `.dbxinit` file (or the `.dbxrc` file if you are using Korn shell) if you want them to appear in each Debugger session.

4.3.3 Executing Commands at Load Time

Because Debugger reads the `.dbxinit` (or `.dbxrc`) initialization file before it loads the program to be analyzed, it will not execute any process control or event management commands it finds in the initialization file. However, you can issue such commands by putting a Debugger `alias` command in the initialization file and have the `alias` command include in its definition the Debugger `source` command. The `source` command instructs Debugger to read and execute, in order, the commands contained in a specified file. You can place any debugging commands you want executed automatically in that file.

4.3.4 Editing Code in the Source Pane

The Debugger source pane functions in many ways like the SourceBrowser source pane: it is read-only until you convert it to a Text Editor pane by choosing Enable Edit from a window menu. Like SourceBrowser, the Enable Edit item enables you to check the file out of SCCS, the version control system; however, you must check the file back into SCCS from a Command Tool or Shell Tool, not from the Debugger.

Starting Another Editor from within Debugger

You can edit the code shown in the source pane with another editor. The Edit item on the Program menu starts your system editor (the editor specified by your SunOS environment variable, `$EDITOR`) in a separate shell and automatically loads the currently displayed source.

4.3.5 Using the Replay Command

One Debugger feature, the Replay command on the Execution menu, is especially valuable for finding elusive bugs in deterministic programs (that is, programs such as compilers that take exactly the same execution path every time they get the same inputs).

4.4 Analyzer

The Analyzer interprets the experiment file created by the Collector and presents the results in a variety of formats. These results furnish information that can be used to improve a program's performance. In cases where paging is a performance bottleneck, the Analyzer can improve performance automatically by instructing the linker to remap functions in memory more efficiently.

Before exploring the capabilities of the Analyzer, reviewing the steps to developing high-performance software might be helpful. The steps are:

1. Base each routine in the program on the most efficient algorithm available.
2. Identify bottlenecks during a typical program run with performance profiling tools.
3. Eliminate the bottlenecks by managing memory more efficiently, streamlining or reducing use of the most frequently called routines, or reducing input/output activity.
4. Recognize that you are done when the program meets its performance objectives.

The Analyzer can help you with steps 2, 3, and 4. Step 1 takes place before coding starts and is not actually part of the performance evaluation process.

4.4.1 Shortcomings of `prof` and `gprof`

The utilities `prof`(1) and its enhanced version, `gprof`(1), construct profiles of time spent per routine, subroutine call, and average time spent in each routine per call. `gprof` also produces a call graph that identifies the run-time relationships between routines. The graph can be used to apportion each routine's call count and time consumption data among its callers. While useful in identifying program bottlenecks, these tools suffer from several major shortcomings, which have been overcome by the tools:

- `prof` and `gprof` require that the program to be profiled be "instrumented" at compile time with extra code that gathers profiling data during a program run.

  In contrast, the performance profiling tools require no special compile flags. Instead, they rely on SunOS 5.0 system calls to provide the necessary information concerning function calls and memory usage. This capability extends to libraries.

- With `prof` and `gprof`, the interval over which each sample of profiling data is accumulated is constant (20 ms) and cannot be changed easily. The developer has no control over the granularity of data collection.

  The profiling tools can collect and display data over any reasonable sampling interval.

- `prof` and `gprof` only present information about user run time. Time spent in other activities (servicing system calls and input/output activities, for example) is not profiled.

  The profiling tools provide information about ten different classifications of process time.

- `prof` and `gprof` collect data over an entire program run — they cannot collect or display data for only a particular part of a run.

  In most cases, only one part of a program needs to be improved. The profiling tools can collect data over any part of a program run and display information on any subset of the collected data.

- `prof` and `gprof`, together with other tools traditionally used in profiling (such as `time`(1V) and `size`(1)), present their output in tabulated text form suitable for display on a terminal. Their output is difficult to interpret quickly.

  The profiling tools take full advantage of the OpenWindows user interface to present data in easily understood graphical form.

The current release has no feature that duplicates the capabilities of `tcov`(1), the utility that exposes statement-level execution frequency for C and FORTRAN programs. The Pascal equivalent of `tcov` is `pxp`(1). These utilities tell you whether or not all branches of a program have been exercised and so are useful for developing test suites and finding sections of dead code.

4.4.2 Analyzing Experiment Data

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

The performance data that you can examine in the Analyzer includes:

- **User time —** Time spent executing program instructions

- **Fault time —** Time required to service fault-driven memory activities, classified into text and data page faults
- **I/O time —** Time the operating system spent waiting for I/O (input/output) operations, such as writing to a disk or tape
- **System time —** Time the operating system spent executing system calls
- **Trap time —** Time spent in executing traps (automatic exceptions or memory faults)
- **Lock wait time —** Time spent waiting for lightweight process locks
- **Sleep time —** Time the program spent sleeping
- **Suspend time —** Time spent suspended (includes time spent in the Debugger during breakpoints and the time used by the Collector to gather data)
- **Idle time —** Time spent waiting to run while the system was busy
- **Function sizes —** Sizes of functions in the program
- **Module sizes —** Sizes of modules in the program
- **Segment sizes —** Sizes of segments in the program
- **Memory usage —** Memory page reference and modification data. Memory pages are characterized in the following ways:
  Modified — A page that is written to
  Referenced —A page that is read from
  Unreferenced —A page that has neither been referenced nor modified
- **Resource usage —** Information about the system resources that are used by the program, including major and minor page faults, process swaps, number of input and output blocks, number of messages sent and received, number of signals handled, number of voluntary and involuntary context switches, number of system calls, number of characters of I/O, and number of working set memory pages

4.4.3 Reordering Program Functions

A program with large memory requirements runs slowly on a machine with limited memory because data and text pages must be swapped in frequently from disk. Although control of data page swapping may be out of the programmer's hands, text page swapping is not: a high number of text page faults is a good indicator that often-used functions are distributed across pages and should be reordered. However, manually reordering a program's functions in source files is difficult because some functions may repeatedly call other functions, obscuring the most efficient ordering from a human observer.

The Analyzer, in cooperation with the compiler and linker, can reorder a program automatically. The program is reordered with a simple but effective algorithm: functions are placed in memory in the order of their frequency of use, the most-used functions grouped together on the same set of pages. As a result, whenever one of these functions is called there is a high likelihood that it will already be in memory.

Automatically reordering the program's functions requires recompilation and relinking. The steps are:

1. Compile the program with the `-xF` option, which causes the compiler to generate functions that can be relocated independently.
2. Run the program under Debugger control and collect profiling data with the Collector.
3. Load the resulting experiment record into the Analyzer, and click on the Create Mapfile button. You will be asked to give the mapfile a name.
4. Create a new, reordered executable with the linker and a special option that uses the mapfile to determine linking order.

You can check to see how much your program's performance has improved by rerunning the experiment on the reordered executable while collecting the same profiling data, loading the resulting experiment record into a second instance of the Analyzer, and comparing the displays with those that resulted from the original run.

4.4.4 Exporting Experiments

You can export the data collected by the Debugger to files for use by other programs such as spreadsheets or custom-written applications. The format of the export data file is documented in *Performance Tuning an Application*.

4.5 FileMerge

The FileMerge tool merges two versions of the same source file, with or without reference to a common earlier version. Several circumstances could make merging source files necessary:

- Two programmers have been assigned to work on the same source file, perhaps in order to kill different bugs. After the work has been done, the files must be merged, compiled, and tested to make sure that the fixes don't interfere with each other.
- A program is being developed primarily for one system but must also be ported to a second system. As bugs are fixed and features added in the primary code, the source must be merged with existing secondary code to produce a program with both the new features and the secondary system calls.
- Bug fixes in a new release must be backported to an earlier release, but new features should not be backported. The responsible developer must examine each difference between source files in the two releases to determine which are bug fixes and which are new features.

4.5.1 FileMerge — `diff` with a Difference

The traditional method of merging two source files is slow and prone to error: you run `diff`(1) on the files and then open one of the files in an editor. By examining the output of `diff` and editing the file, you can eventually construct a merged version.

In contrast, FileMerge presents the two files to be merged in side-by-side text panes for easy comparison. It clearly marks the differences between the files and automatically constructs an output file.

4.5.2 FileMerge Window

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

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

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

When two files have been loaded without an ancestor file, FileMerge does not mark additions and deletions in the input files because it has no reference to determine whether a line has been added to one file or deleted from the other.

4.5.3 Merging Files

By default, FileMerge places all lines that are common between the two input files into the merged output file. When a line differs between the two files, you can decide which of the two lines to place into the output file. Each time you resolve a difference, FileMerge advances automatically to the next unresolved difference. If neither input file contains a suitable line to use in the output file, you can edit the output file directly. See *Merging Source Files* for details.

4.5.4 Viewing Differences Read-Only

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

4.5.5 Loading Lists of Files

You can specify lists of files to load sequentially if you start FileMerge from the command-line interface. This capability is very useful when two entire directories must be merged. You can also specify a list of ancestor files at startup. Note that you can customize the Manager (through its Properties windows) to start FileMerge with any command-line options you desire, including loading files from a list.

Each file in a pair (or triple, if you are also loading an ancestor file) to be merged must have the same local name. The files themselves must be stored in separate directories so that their absolute path names are different. You then create a list of file names in a *listfile* and start FileMerge with the `-l` option while specifying the name of the listfile, the two directories to be merged, an output directory, and optionally a third directory of ancestor files. You can then automatically load files successively as you finish resolving differences in each file pair.

In the example shown in Figure 4-1, Bill and Lucy must merge their files named `A`, `B`, and `C`. The version of the files they started with is in the directory `/usr/src/ancestor.` Bill (or perhaps Lucy) first creates a list file named `sourcelist` and places it in the ancestor directory (although the list file could be stored anywhere). The content of the file is the local names of the files to be merged: `A`, `B`, and `C`. Bill then starts FileMerge with the following shell command:

```
filemerge -a /usr/src/ancestor -f1 lucy -f2 bill
-l /usr/src/ancestor/sourcelist
/usr/src/lucy /usr/src/bill /usr/src/output &
```

The shell interprets the command options as follows:

- The `-a` option specifies the name of the ancestor file directory.
- The `-f1` and `-f2` options specify the names that will be displayed over the right and left text panes, respectively. The names provide a convenient reference to keep track of which files are displayed in which pane.
- The `-l` option specifies the name of the list file.
- The three final directory names specify the left, right, and output directories, respectively.
- The ampersand (&) causes FileMerge to run in background mode.

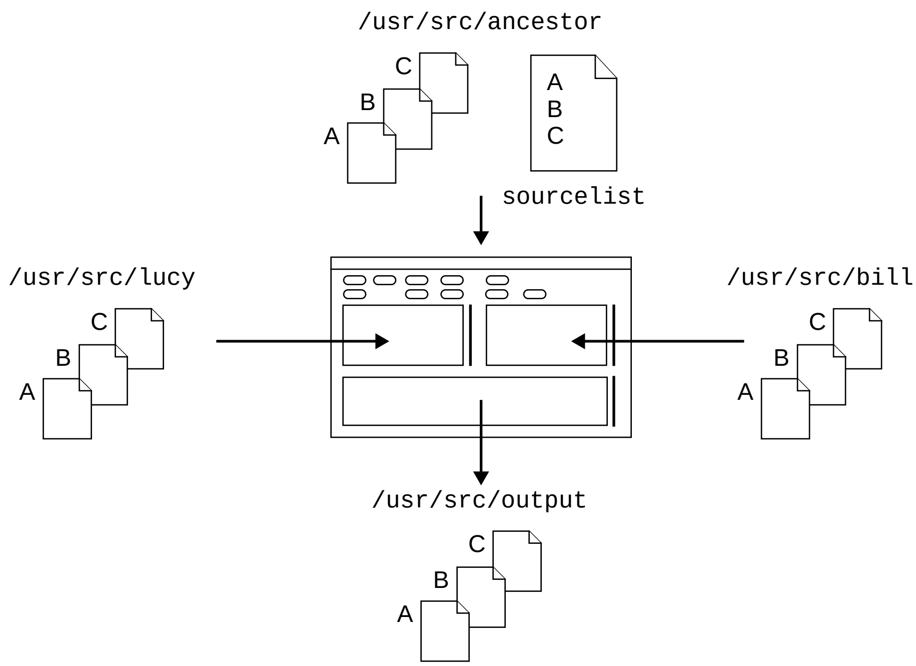


Figure 4-1 Loading Files Successively from a List

FileMerge starts up and loads the file named A from each of the three directories, displaying Lucy's file in the left pane and Bill's in the right. When all differences between the files have been resolved, Bill clicks on the Save button to store A in the output directory. Bill then chooses the Load Next From List item from the File button menu to load file B and repeats the process until he has merged all files named in the list file.

More About the Debugger A

This appendix provides a brief description of the debugging features available in the Debugger and dbxtool.

A.1 Debugger Features

Displaying the Contents of Memory

You can display the contents of memory in several ways. The simplest way is to print the address of a single variable or pointer. You can also print the contents of memory locations in the following ways:

- From a starting point address through some specified number of increments of the starting point addresses, *counting* from the starting point.
- From a starting point address through a second, endpoint address.
- From the contents of the next address after the one most recently displayed; and also (optionally), a specified number of succeeding addresses.

Displaying Source Code

Like any good source-level debugger, the Debugger displays source or assembler code corresponding to the machine code being executed. You can also “visit” source files other than the file that corresponds to the program currently running.

Single-Stepping through the Program

You can single-step through the program by machine instructions or by source lines. You can decide to step into procedures (with the `step` command) or over them (with the `next` command). Note, however, that single-step commands do not work with optimized code (programs compiled with the `-g -O` options).

In a stopped program, you can specify a line at which to resume program execution (with the `cont at` command). This feature enables you to skip over one or more faulty lines of code without having to recompile.

Checking for Runtime Errors

One of the first debugging tasks is looking for and fixing runtime errors. With the Debugger's Runtime Checking (RTC) feature, you can automatically detect runtime errors during the development phase. Compiling with the `-g` flag provides source line number correlation in RTC's error messages. RTC can also check programs compiled with the optimization `-O` flag.

Note – This feature available with SPARCworks/ProWorks running Solaris 2.x.

Examining Variables

With the Visual Data Inspector (VDI), you can examine program variables including complex structures and monitor values during program execution. You can also inspect two-dimensional arrays in a spreadsheet-like format.

Note – This feature available with SPARCworks/ProWorks running Solaris 2.x.

Setting Breakpoints

You can set breakpoints at a source line, instruction address, procedure, or function. Breakpoints (and tracepoints) can be listed and cleared as a group or individually. You can set multiple breakpoints in C++ member functions by setting breakpoints in functions of a class.

You can also set conditional breakpoints (sometimes called watchpoints) on variables, functions, source lines, or expressions you specify. Conditional breakpoints cause the Debugger to stop a program (or perform other actions) only if a specified condition becomes true or the value of a specified variable changes.

This feature makes use of the `when` command, which tests for the truth of a condition before executing the command you specify. When the condition is met, the Debugger can execute a `dbx` command (which could be other than a `stop` command) or evaluate any expression you have specified. It is especially useful when you want to execute a command that does not stop program execution when the condition has been met. Contrast this feature with post-event condition testing, discussed under "Setting Postbreak Modifiers."

Setting Tracepoints

You can set tracepoints for variables, lines, and expressions in lines. You can set bounded traces for all statements in functions, member functions, classes, or entire programs. When the Debugger reaches the tracepoint, it performs any `trace` subcommand you have specified, such as echoing the identified code as it executes or printing the line numbers of statements, the values of variables or arguments passed to functions, or the names of member functions.

Setting Postbreak Modifiers

You can set post-break conditional modifiers for breakpoints and tracepoints. A postbreak modifier instructs the Debugger to test for a condition after the program arrives at a tracepoint or stops at a breakpoint. If the post-event condition is true, the Debugger stops the program or begins tracing program execution; if it is false, the Debugger continues program execution. The Debugger implements this feature with the `stop` or `trace` command, followed by one or more `if` clauses.

Managing Events

You can use the Debugger's event management commands to create and manipulate event handlers so that when specific events in the program occur, certain operations are performed in the program being debugged. Event management is an enhancement to setting breakpoints and traces.

Navigating the Call Stack

You can move up and down the call stack (sometimes called *walking the stack*). The call stack represents all currently active routines; that is, routines that have been called but have not yet returned to their respective callers.

Debugger commands can list all active routines, move up the stack to the routine that called the current routine, or move down the stack to routines that were called later.

The Stack command is especially useful for learning about the state of a program that has failed and produced a core file. When a program fails and produces a core file, you can load the core file into Debugger and examine the state of the call stack at the time the program failed. As you debug the core file, you can also evaluate variables and expressions to see what values they had at the time the program crashed.

Inspecting the Call Stack

You can use the Stack Inspector to view the call stack, to hide or show selected functions, and to move from one frame to another. When you visit another process or thread, the Stack Inspector automatically updates the stack.

Handling Overloaded Names

When variables from different functions or procedures have the same name, you can differentiate each name by fully qualifying it with the name of the function (and source file, if necessary) in which it occurs. If you specify an ambiguous or overloaded name to the Debugger, a popup window opens with a list of fully qualified names from which you can select the correct one.

Fixing Source Code and Continuing Program Execution

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

Calling Functions

When a program is stopped, you can call a function with the `call` command. The command accepts values for the parameters that must be passed to the called function and runs the function. If the function was compiled for debugging, the Debugger checks to see that you are passing the correct number of arguments. The Debugger honors any breakpoints that you have set in the function.

To run a function and print its return value, type the following command in the command pane.

```
(debugger) print function_name
```

Analyzing Live Processes

You can attach the Debugger to an executing program such as a daemon, debug the process, and then detach the Debugger. To attach the Debugger to a process, start the Debugger with the process ID of the program as an argument; to detach the Debugger, simply issue the Debugger `detach` command with the same process ID as argument.

Handling Signals

You can intercept signals and act on them, then continue execution even when signals have been received that would ordinarily force a halt. You can also specify a system signal and resume execution; the program will behave as if it had received the signal normally.

Inspecting Threads

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

Note – This feature available with SPARCworks/ProWorks running Solaris 2.x.

Analyzing Dynamically Linked Libraries

The Debugger provides full debugging support for programs that use dynamically linked, shared libraries. In contrast with statically linked libraries, which are linked when the program executable is first created, dynamically linked libraries are loaded in one of two ways:

- Automatically when the executable is started.
- On demand with calls to `dlopen()` and related functions while the program is running. Debugging support for `dlopen()/dlclose()` enables you to step into a function or set a breakpoint in functions in a dynamically linked library just as you can in a library that links at start-up.

In either case, the run-time linker binds and unbinds dynamically linked libraries during program execution.

For either statically or dynamically linked libraries, full debugging support depends on the libraries having been compiled using the `-g` option.

Analyzing Multiple Languages

The Debugger works with the same SPARCompilers as SourceBrowser. At this writing, they are C, C++, FORTRAN, and Pascal. The Debugger can analyze programs with sources made up of combinations of these languages.

Reading Symbol-Table Information on Demand

Reading symbol-table information on demand speeds up loading the Debugger because the entire compiled symbol table does not need to be read at startup. This feature is especially appreciated by developers who work with very large programs.

Note that symbol table information is included by default in intermediate `.o` files when you compile for debugging (that is, with the `-g` option). This arrangement has the advantage of producing relatively small executables, but it also means that you cannot routinely destroy the `.o` files until debugging is finished.

Intermediate `.o` files that are associated with shared libraries (that is, that are used to build `lib.so` modules) must also be preserved. These libraries typically include, as a minimum, `libtt.so` and `libX11.so`. If the conditions at your site are such that keeping `.o` files for shared libraries is not practical, you can place the symbol tables in the executable itself by compiling with the `-xs` option. However, in this case the entire symbol table will be read when you start the Debugger, increasing start-up time.

Analyzing Optimized Code

Unlike `dbxtool`, the Debugger handles optimized code; that is, code compiled with both the `-O` and `-g` options. When analyzing optimized code, you can stop execution at the start of any function and display global variables and arguments.

Some restrictions apply when working with optimized code: you cannot use the single-stepping commands `next` and `step`, and you cannot evaluate or assign values to local variables and arguments.

Saving, Restoring, and Replaying Sequences of Commands

You can save analysis sessions in the form of command sequences. The sequence can then be replayed to restore the analysis session, or a subset of the sequence can be replayed for an “undo” effect (on programs that are deterministic).

Stepping out of Functions with a Single Command

After you have stepped into a function, a single command (`step up`) enables you to quickly return to the calling function.

Monitoring the Values of Expressions and Variables

A Data Display popup window monitors the values of expressions and variables (including the values of nested pointers) whenever the program stops. You can also spot-check the value of selected expressions with `print` commands.

Assigning Values to Variables

You can stop the program and change the values of variables. When used with the `cont at` (continue at) command, this feature enables you to assign known values to variables, skip over faulty code, and continue program execution.

Evaluating Arrays

You can evaluate arrays the same way you evaluate other types of variables: select the array, then choose Evaluate from the Data menu.

For FORTRAN arrays, use the Debugger `print` command to evaluate part of a large array. FORTRAN array evaluation includes:

- **Array Slicing** — Print any rectangular, *n*-dimensional subset of adjacent elements in a multidimensional array.
- **Array Striding** — Print certain elements only, in a fixed pattern (for example, every third element), within a slice or an entire array.

You can slice an array with or without striding. The default stride value is 1, which prints each element.

Checking Out Source from SCCS

Like the source pane in SourceBrowser, the source pane in the Debugger can be enabled as an OpenWindows editing window. If the source file is controlled by SCCS, the Debugger can check out the file directly in the editing window. The edited file can be saved in the Debugger, but SCCS files must be checked back in with another tool or from a shell command line.

Displaying Command History

The Debugger records each command you enter. You can display the history of a session by typing `history` in the command pane. You can also issue a command from the list by number, using substitution commands, which follow the C-shell `history` conventions (`!!`, `!`*n*, `!`*chars*).

Programmability

The Debugger command language is based on the syntax of the Korn shell, including I/O redirection, loops, built-in arithmetic, history, and command-line editing (only in command-line mode, not available from the Debugger).

Index

N

O