Power Mac Debugger Reference

For version 2.1

Contents

Figures, Tables, and Listings

About This Book

This book describes how you can use the Power Mac Debugger, version 2.1, to debug your programs and measure their performance. It is intended for developers creating applications, libraries, or other programs for PowerPC™-based Mac OS computers.

What's in This Book

This book contains several chapters and appendixes. Chapters 1 and 2 show how to install the Power Mac Debugger and get set up for debugging. Chapters 3 and 4 provide basic information about using the debugger to debug your code. The remaining chapters and appendixes discuss additional topics that will be of interest to developers with specific needs. Appendix E provides a quick reference guide that you can refer to once you are familiar with the debugger.

- Chapter 1, "Introduction," describes the architecture of the Power Mac Debugger and shows you how to install it on your system.
- Chapter 2, "Getting Started," shows you how to build your code for debugging, launch the Power Mac debugger, and target your code.
- Chapter 3, "The Debugger Windows," describes the appearance and functionality of each of the debugger's windows.
- Chapter 4, "Basic Debugging Tasks," explains how to accomplish the most common debugging tasks, such as setting breakpoints, stepping through your code, and examining variables.
- Chapter 5, "Advanced Debugging," discusses some specialized topics, such as debugging non-application code.
- Chapter 6, "Measuring Performance," shows how to use the Adaptive Sampling Profiler to help you tune your application's performance.
- Chapter 7, "Troubleshooting," contains solutions to the most common difficulties people can have when using the debugger.

- Appendix A, “Debugger Preferences,” discusses the many options you can set to customize the debugger.
- Appendix B, “Expression Evaluation,” documents the grammar of C and C++ expressions that the debugger can evaluate.
- Appendix C, “Debugger Extensions,” describes how you can create debugger extensions to customize and extend debugger functionality.
- Appendix D, “Creating Custom Unmangle Schemes,” shows how to use a custom C++ unmangling scheme with the debugger.
- Appendix E, “Quick Reference Guide,” has two tables summarizing the most common tasks you can perform with the debugger and the windows you work in.

Related Documentation

This books assumes basic knowledge of PowerPC-based software development on the Mac OS platform. For further information, the following books may be helpful:

- *Mac OS Runtime Architectures*
- *Inside Macintosh: PowerPC System Software*
- *Macsbug Reference and Debugging Guide*
- *Building and Managing Programs in MPW*, second edition

Conventions Used in This Book

This book uses various typographic conventions to present certain types of information. Words that require special treatment appear in specific fonts or font styles.

Special Fonts and Font Styles

`Letter Gothic`	Throughout this book, words that you must type exactly as shown are in Letter Gothic.
	Letter Gothic font is also used in text for command names, command parameters, arguments, and options, and for resource types.
boldface	Key terms or concepts are shown in boldface and are defined in the glossary.

Syntax Notation

The following syntax conventions are used in this book:

`literal`	Letter Gothic text indicates a word that must appear exactly as shown. Special symbols (∂, •, §, &, and so on) must also be entered exactly as shown.
italics	Italics indicate a parameter that you must replace with anything that matches the parameter's definition.
[ ]	Brackets indicate that the enclosed item is optional.
. . .	Ellipsis points (. . .) indicate that the preceding item can be repeated one or more times.
|	A vertical bar (|) indicates an either / or choice.

Types of Notes

This book uses two types of notes.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. ▲

Code Samples

All code listings in this book are shown in C or C++ (except for listings that describe resources, which are shown in Rez-input format). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application. You can find the location of this book's code listings in the list of figures, tables, and listings.

To make the code listings in this book more readable, only limited error handling is shown. You need to develop your own techniques for detecting and handling errors.

This book occasionally illustrates concepts by reference to sample applications called *CTubeTest, DemoDialogs,* and *SortPicts.* These are not actual products of Apple Computer, Inc.

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog,* contact

Apple Developer Catalog
Apple Computer, Inc.

P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
Internet	order.adc@apple.com http://www.devcatalog.apple.com

Introduction

Contents

The Power Mac Debugger is an application that allows you to debug software written for PowerPC™-based Mac OS computers at a source-code or assembly-language level. You can use the Power Mac Debugger to debug applications, shared libraries, stand-alone code and system software. You can run the Power Mac Debugger on the machine containing the code you are debugging or on a different machine.

First, this chapter describes the architecture and basic capabilities of the debugger. Then it explains how to install the debugger on your system. Finally, it describes how to connect two machines if you are using a two-machine debugging setup.

About the Power Mac Debugger

The purpose of a debugger is to help you find errors in your programs. To do so, it allows you to control the execution of a program and examine its internal state at various points.

The Power Mac Debugger allows you to do **source-level debugging**. For most programmers, a source-level debugger makes debugging easier and quicker. It allows you to see your source code as you debug. With it, you can easily browse and display your code. In addition, you can stop the program at any line of code and examine the values of your variables.

Source-level debugging is made possible by the use of **symbolic information files**, or **symbol files**, for short. Symbol files tell you which statements of your program correspond to which machine instructions in your executable code. They also tell you where your variables reside in memory. If you do not provide a symbol file, the Power Mac Debugger still allows you the option of doing **assembly-level debugging**. You can examine memory and registers, and disassemble any area of memory to machine instructions.

The Power Mac Debugger consists of a series of windows that display information and commands to control the execution of your program.

The debugger lets you control the execution of a program in various ways:

- You can set **breakpoints**, which cause the debugger to stop at a particular statement in your program. Breakpoints can have conditions attached to them; for example, you can stop at a breakpoint only when a certain variable has a particular value.

- You can **single-step** through your code, that is, execute one instruction at a time. When stepping, you have the option to step over or into subroutines.

When the program is stopped, you can

- display the stack and navigate through routines in its calling chain
- display and edit local and global variables
- display and edit general-purpose and floating-point registers
- display and edit any range of memory
- disassemble PowerPC and 680x0 code

In addition, you can

- display and individually target the processes and threads running on your machine.
- measure the performance of a program
- evaluate C and C++ expressions
- customize or extend the capabilities of the debugger by writing debugger extensions (similar to `'dcmd'` resources used with the MacsBug debugger)

The Host-Nub Architecture

The Power Mac Debugger has a flexible architecture. It allows you to debug using a single machine or two machines and provides different capabilities for different debugging needs.

Here are some terms that are used throughout this book:

- The code you debug is called the **target**. It can be, for example, an application, a driver, or a shared library.
- The application you interact with when you debug is known as the **host**. It has a user interface, with several windows that display information about the target and commands to control the execution of the target. It can be running on the same machine as the target or on another machine connected to the target machine.
- The **nub** is software that runs on the target machine and actually controls the code being debugged. The host communicates with the nub, not directly with the target code. You do not interact directly with the nub. The nub can handle exceptions, set breakpoints, single-step through the target code, and

read and write memory and registers. When the target is stopped (for instance, at a breakpoint) the nub provides information to the host about the state of the machine, and the host displays this information to the user. The Power Mac Debugger provides two different nubs that can be used in different situations. See “The Nub Software” (page 23) for more details.

There are several advantages to having a separate host and nub:

- The user interface is consistent, regardless of the type of software being debugged. Whether you are debugging applications, extensions, or drivers, or are working at the source level or assembly level, you use the same host application. You can use different nubs depending on the type of debugging you are doing.
- The host and nub can be on a single machine, or they can be on separate machines that communicate with each other. Each of these setups has advantages and disadvantages:
    - Single-machine debugging eliminates the need for additional hardware. It can also be more convenient as you make changes to your program, since you can have your development environment and the debugger host and nub on the same machine. (With a two-machine setup, your source files and symbol file are on the host machine. As a result, you need to move the executable code to the target machine.)
    - Two-machine debugging prevents the debugger host from interfering with the target. This setup is essential in low-level debugging. It is also advantageous when you are debugging code that draws to the screen, since the host’s windows may make it difficult to see what is happening in the target application. In addition, the target cannot interfere with the host; if the target machine crashes, the host machine (which usually includes the development environment as well as the debugger) is still running.

The Power Mac Debugger, version 2.1, provides two nubs. The one you use depends on the type of code you are debugging, the system it is running on, and the level of control needed on the target machine. The next section describes these nubs.

The Nub Software

There are two nubs that can be used with the Power Mac Debugger:

- **The application nub** (also known as the high-level nub). This nub runs as a background application on the target machine. When you are debugging, the nub takes control of the target process, but other applications continue to run. The host may run on the same machine as the nub or on another machine connected to the target machine.
- **The serial nub** (also known as the low-level nub). This nub takes control of the target machine when the target code is stopped. In effect, the entire machine is stopped. No other applications can run, and file servers are likely to time out. Using the serial nub requires two machines (connected via the serial port), since the host cannot run on the same machine if the machine is stopped.

Most developers use the application nub for their debugging. You *must* use it if you are debugging code that uses cooperative threads under the Thread Manager; the serial nub works with programs on a process basis and is not aware of individual threads.

The serial nub is useful for debugging system code, drivers or other low-level code. You *must* use it if you want to debug code that runs at interrupt time.

Installation

This section describes how to install the debugger software; it shows you which files are needed and where they should be installed. Some files are put on the host machine, others on the target machine. In a single-machine configuration, the host and target are the same machine.

Once you've installed the debugger software, Chapter 2, "Getting Started," shows you how to launch the debugger and target your code.

Installing the Debugger Host

The Power Mac Debugger is the host application you interact with when you are debugging any Mac OS software. This application takes up approximately 4 MB of disk space and requires a minimum of 5 MB of RAM to execute (it takes an extra 2 MB if virtual memory is off). It is a "fat" application; that is, it can run on either a PowerPC-based Mac OS computer or (as the host in a two-machine setup) a 680x0-based system. System 7.1.2 or later is required for a PowerPC-based Mac OS system and 7.1 or later for a 680x0-based Mac OS system.

To install it, drag the Power Mac Debugger application to any location on the host machine.

Installing the Application Nub

The application nub is contained in a file called Power Mac DebugServices. It is compatible with system software version 7.5.1 or later. It runs as a faceless background application on the target machine and uses about 200 KB of RAM. To install it, drag Power Mac DebugServices to the hard disk of your target machine.

When you launch the Power Mac Debugger on a single-machine setup, the host automatically launches the application nub. With two machines, you must be sure the application nub is running in order to debug. You can have the application nub launch automatically when you start the target machine by installing it in the target machine's system Startup Items folder.

Note
The application nub does not appear as an item in the Application menu. If you want to quit the nub when you've finished debugging, press the keys Shift-Control-Delete and hold them down until you see an alert with the message "Quitting DebugServices". ◆

Installing the Serial Nub

The serial nub stops the target machine whenever a breakpoint or other exception occurs in the target application. For that reason, it must be installed on a different machine from the host. To install it, perform these steps:

1. **Drag the files PPC Debugger Nub and PPCDebuggerNubINIT onto the System folder icon on the target machine.**

   These files are installed in the Extensions folder and contain the actual serial nub code.

2. **Drag the file Debugger Nub Controls onto the System folder icon on the target machine.**

This file is installed in the Control Panels folder. It is used to determine which serial port is used to connect the two machines and whether the nub is active. See "Using Debugger Nub Controls" (page 29) for information on using this control panel.

3. **Restart the target machine so that you can begin debugging.**

Note
You may want to have the serial nub and application nub installed on the same target machine and use them at different times. To use the application nub, the serial nub must be disabled using Debugger Nub Controls; see "Using Debugger Nub Controls" (page 29). ◆

Installing Optional Files

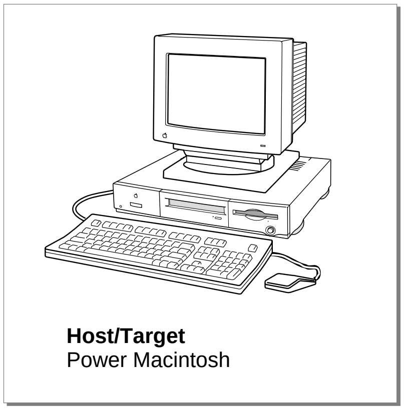
You may wish to install additional software to aid your debugging efforts:

- **MacsBug.** You can use the Power Mac Debugger without MacsBug, but there are some advantages to having MacsBug installed. It allows you to set breakpoints and step through emulated 680x0 code, and has commands for examining various system data structures. Install MacsBug in the System Folder on the target machine. See Chapter 5, "Advanced Debugging," for more information on using MacsBug with the Power Mac Debugger.
- **A ROM map,** which contains information about symbols in ROM code. A ROM map can be useful when disassembling ROM code or generating a performance report. To install it, drag the RomInfo file to the same location as the Power Mac Debugger.
- **Preferences files.** If you have written debugger extensions or custom C++ unmangle schemes, you need to create preferences files for them. See Appendix C, "Debugger Extensions," and Appendix D, "Creating Custom Unmangle Schemes," for information on these.

Connecting the Host and Target Machines

When debugging on a single machine, the debugger host and nub, as well as the target software, reside on the same machine, as shown in Figure 1-1.

Figure 1-1 Single-machine debugging



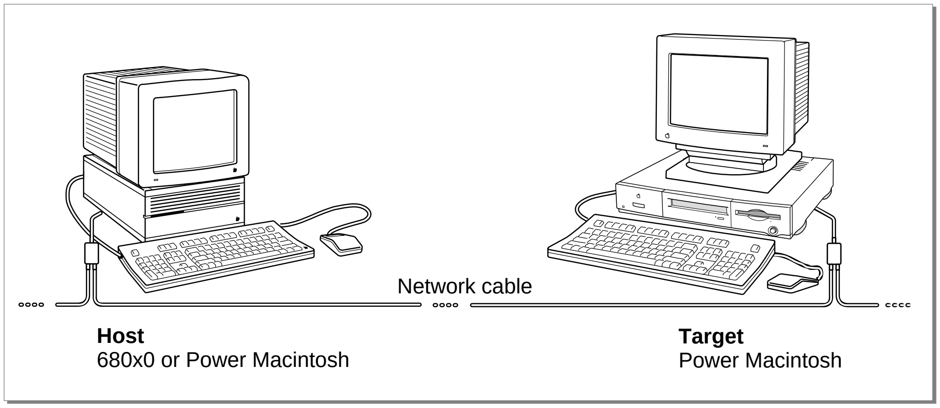
In contrast, when using two machines, the software you are debugging runs on one machine (the target machine), while the debugger host runs on another machine (the host machine). The host machine can be any Mac OS computer with a PowerPC processor or a 68020 (or later) processor. (For the most efficient processing, a Macintosh Quadra or PowerPC-based Mac OS computer is recommended.)

How you connect the two machines depends on whether you are using the application nub or the serial nub. These two ways are described next.

Connecting When Using the Application Nub

If you are using the application nub, you connect the two machines through an AppleTalk network connection, as shown in Figure 1-2. The host communicates with the nub through the Program-to-Program Communication (PPC) Toolbox. You must enable program linking on the target machine through the control panels Sharing Setup and Users & Group. For more information, see Chapter 7, “Troubleshooting.”

Figure 1-2 Two-machine debugging using the application nub



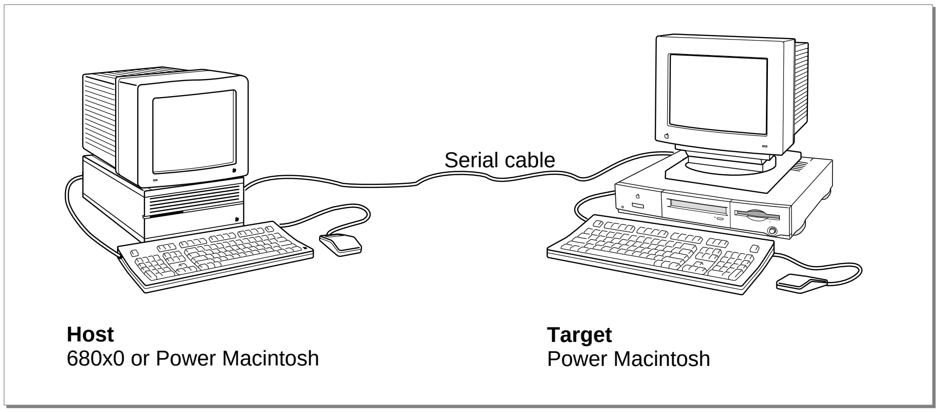
Connecting When Using the Serial Nub

If you are using the serial nub, you connect the machines through one of the serial ports, as shown in Figure 1-3. To indicate to the nub which port you are using, use the Debugger Nub Controls panel on the target machine. The next section provides more details on using this control panel.

IMPORTANT
For best results, use the System Peripheral 8 Cable (part number M0197LL/B) to connect the host and target; other cables may result in communications failures. If possible, avoid running the cable near monitors or power cables. ▲

For best results, unmount all server volumes before beginning system-level debugging. When the target is stopped, the nub disables interrupts, preventing continuous communication between your computer and file servers. As a result, you may experience delays while your computer waits for connections with a server to be timed out.

Figure 1-3 Two-machine debugging using the serial nub

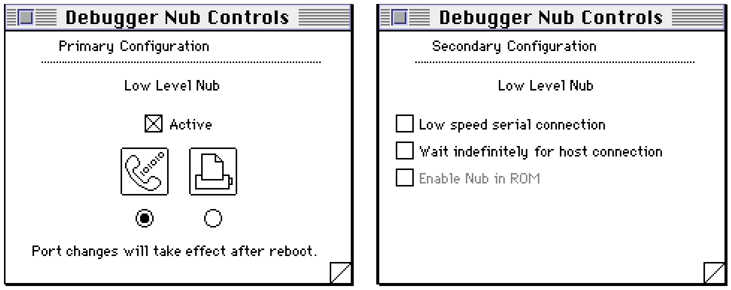


Using Debugger Nub Controls

When you use the serial nub, you can set several options by opening the Debugger Nub Controls control panel on the target machine (see the left window in Figure 1-4). You can specify

- whether the nub is active. It is possible to use the application nub and the serial nub at different times, but you must make the serial nub inactive if you want to use the application nub.
- which serial port you are using to connect to the host machine.

Figure 1-4 The Debugger Nub Controls windows



Debugger Nub Controls

Primary Configuration

Low Level Nub

[✓ Active]

Port changes will take effect after reboot.

Debugger Nub Controls

Secondary Configuration

Low Level Nub

☐ Low speed serial connection
☐ Wait indefinitely for host connection
☐ Enable Nub in ROM

You can set additional options by clicking the page-turning control in the lower-right corner of the window. These options are:

- **Low-speed serial connection.** Check this box if you are using a host computer that is slower than a Macintosh Quadra and you are having problems maintaining the connection between the host and target machines. You must set the same preference on the host machine, using General Preferences.
- **Wait indefinitely for host connection.** Unless you check this box, the debugger nub times out after attempting to connect to the host for 5 seconds.
- **Enable Nub in ROM**. This is an option only on certain models of Power Macintosh computers that have PCI buses and that were built with a serial nub in their ROM. On those machines, the option is turned off by default but can be enabled if the user desires. Since this nub loads early in the boot process, it is useful for developers of PCI native device drivers.

C H A P T E R 2

Getting Started

Contents

This chapter shows you how to get up and running with the Power Mac Debugger. It shows you

- how to build your code for debugging
- how to launch the Power Mac Debugger
- how to target your code so you can begin debugging

Building Your Code for Debugging

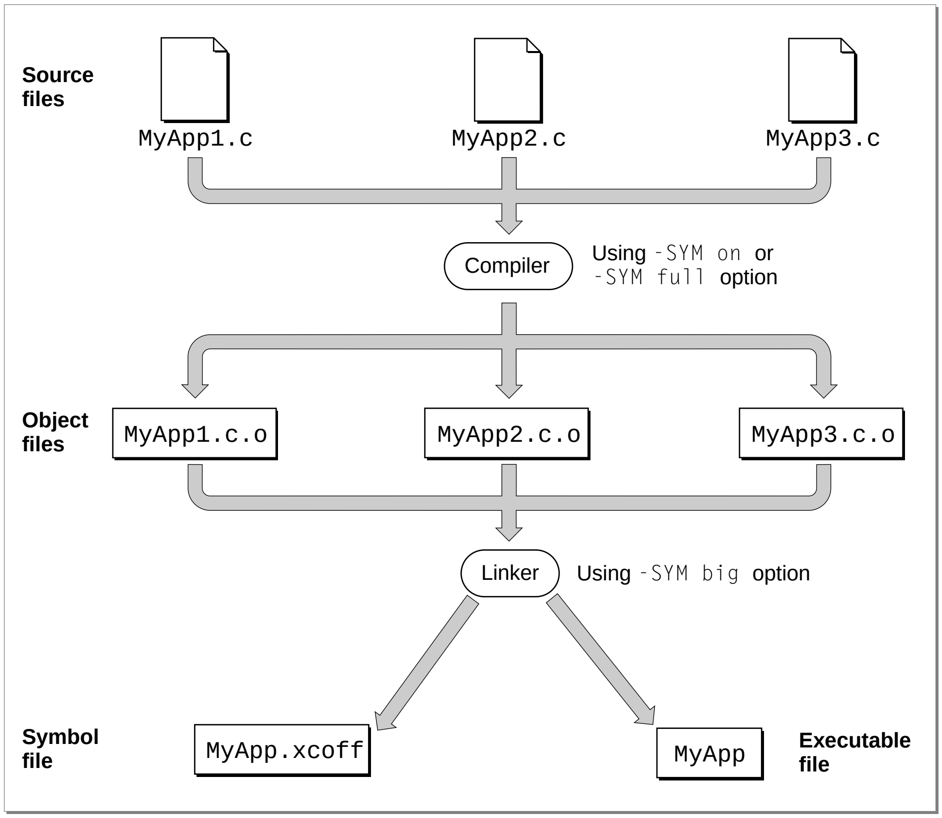
To debug a program at the source-code level, the debugger needs information to associate the statements in your source code with the instructions in an executing program. This information is contained in a symbol file. The Power Mac Debugger can read two types of symbol file formats, those with `.xcoff` and `.xSYM` suffixes.

When you build your code for debugging, you give directions to the compiler and linker to generate the symbol file and the executable program. Make sure to turn off all optimization when compiling. If you generate optimized code, you cannot do accurate source-level debugging.

Figure 2-1 shows how building your application for source-level debugging relates to the build process as a whole.

Note
Normally your symbol file has the same name as your application (or other code) with the `.xcoff` or `.xSYM` suffix added. In certain cases, the names may be different, and you may need to tell the debugger which code goes with a particular symbol file. See “Mapping Symbols To Code” (page 41) for more information. ◆

Figure 2-1 Building your application for debugging (MPW)



IMPORTANT

For debugging on a single machine, applications must have a `'SIZE'` resource (with resource ID -1) with its `CanBackground` bit set to `TRUE`. ▲

Building With MPW

When using an MPW compiler (MrC or MrCpp) to produce native PowerPC code, specify the `-sym on` or `-sym full` options in order to generate symbolic information. The compiler does not produced optimized code if you use these options.

When linking with PPCLink, you generate the symbol file by specifying the `-sym big` option. The output of PPCLink is an `.xcoff` file that is usable by the Power Mac Debugger.

Note
The `-sym on` linker option also produces symbols; however, it is much slower, in both linker speed and variable evaluation in the debugger, and is not recommended. ◆

You can also use the MakeSYM tool (with the `-sym big` option) to create an `.xSYM` file from the `.xcoff` file. Although MakeSYM is not needed when using the Power Mac Debugger, it gives you the option of using some third-party debuggers to debug the code as well.

Building With Third-Party Development Environments

You can use the Power Mac Debugger to debug programs built with any development environment that produces `.xSYM` or `.xcoff` files. Metrowerks CodeWarrior, for example, generates `.xSYM` files when building native PowerPC programs. Consult your development system's manual for information.

Where to Keep Your Files For Debugging

In addition to the debugger itself, the host machine must contain the following files:

- Your `.xcoff` or `.xSYM` symbol file

  If you are doing single-machine debugging, it is recommended that you place the symbol file in the same directory as the target application. The debugger can then easily find the symbol file if you launch the application under debugger control.

- Your source files

  The debugger looks for these files in the same directory in which the symbol file is located. If it can't find them, it prompts you to locate them.

The application you are debugging must be on the target machine.

Launching the Debugger and Targeting Code

This section discusses how to get the debugger running and targeting the code you want to debug. There are several possible sequences you can follow, each one involving a number of steps. To debug, you must

- launch the debugger
- tell the debugger whether you are using one machine or two and, if two, how the machines are connected
- make sure your code is running on the target machine, then stop it under debugger control

To do source-level debugging, you must

- open one or more symbol files
- map the symbol files to code running on the target machine

The following sections discuss these steps in detail.

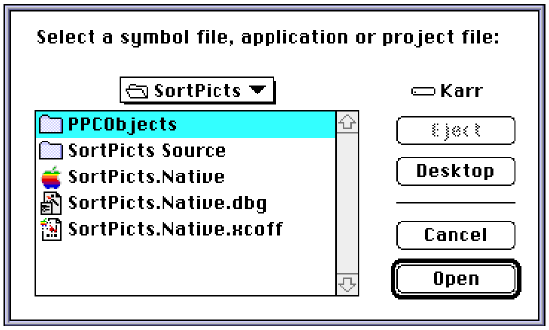
Launching the Debugger and Opening a Symbol File

To launch the Power Mac Debugger, follow these steps.

1. **Double-click the Power Mac Debugger icon.**

   The debugger launches and prompts you for a symbol file to open, displaying a dialog box like the one in Figure 2-2.

Figure 2-2 Opening a symbol file



2. Select a symbol file (`.xSYM` or `.xcoff`) from the dialog box.

The debugger then displays a Browser window for the symbol file. See "The Browser Window" (page 49) for more information on the Browser.

3. Click open.

Instead of selecting a symbol file, you can also do one of these things:

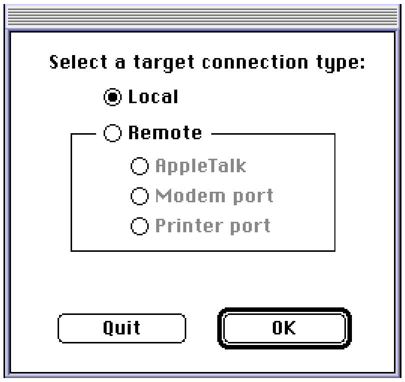
- select a debugger project (`.dbg`) file. The debugger opens the corresponding symbol file. See Appendix A (page 154) for more information on `.dbg` files.
- select an application (single-machine debugging only). The debugger opens the symbol file associated with the application, then launches the application and stops it before it reaches its `main` routine. See "Launching the Target Application" (page 39) for more information about this break on launch procedure.
- click Cancel. In this case, the debugger opens no symbol file. You can open one later by choosing Open from the File menu, or you can do assembly-level debugging without the use of a symbol file.

You can also open a symbol file (an `.xcoff` or `.xSYM` file) directly from the Finder or drag the symbol file's icon onto the Power Mac Debugger's icon. The debugger is launched, if it is not already open. When you open a symbol file this way, the dialog box is bypassed.

Specifying the Target Connection Type

Once you have opened a symbol file, a dialog box (Figure 2-3) appears, prompting you to choose how you wish to connect to the target.

Figure 2-3 Selecting the target connection


Select a target connection type:

(● Local)

○ Remote

○ AppleTalk

○ Modem port

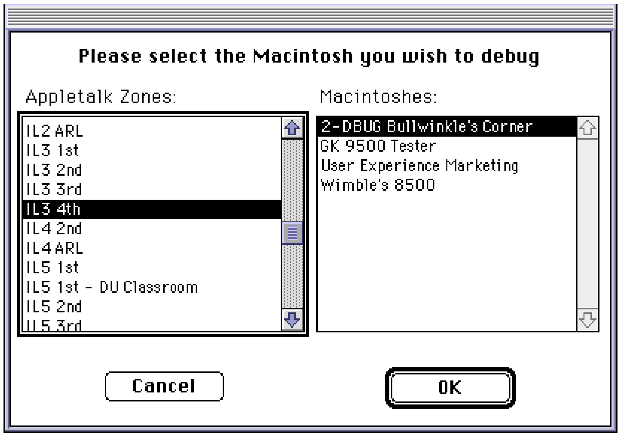
○ Printer port

- If you are debugging on a single machine, click Local, then click OK.
- If you are debugging on two machines, click Remote. Then select the type of remote connection you want.
  - □ If you are connecting to the application nub, click AppleTalk, then click OK. Make sure that both machines are connected to an AppleTalk network.
  - □ If you are connecting to the serial nub, click Modem port or Printer port, then click OK. Make sure the host machine has a serial cable in that port and that the cable is connected to a serial port on the target machine. See "Using Debugger Nub Controls" (page 29) for information about setting the port on the target machine.

Note
If you want to skip this dialog box in subsequent debugging sessions, you can choose Connection Preferences from the Edit menu. A similar dialog box appears, asking you to specify your default setting. From then on, you will not be prompted for your target connection type when you launch the debugger. ◆

If you connected via AppleTalk, a dialog box like the one in Figure 2-4 displays a list of machines running the application nub. Select the one you want and click OK. If you have problems, make sure that both machines are connected to an AppleTalk network. Use the Sharing Setup control panel to make sure program linking is enabled for the target machine.

Figure 2-4 Selecting a target machine


Please select the Macintosh you wish to debug

Appletalk Zones:
IL2 ARL
IL3 1st
IL3 2nd
IL3 3rd
IL3 4th
IL4 2nd
IL4 ARL
IL5 1st
IL5 1st - DU Classroom
IL5 2nd
IL5 3rd

Macintoshes:
2-DBUG Bullwinkle's Corner
GK 9500 Tester
User Experience Marketing
Wimble's 8500

Launching the Target Application

If an application you want to debug is not running, you must launch it.

- **Double-click the application's Finder icon while holding down the Control key.**

  This causes a **break on launch**: the application launches and stops before execution reaches its `main` function. (Be sure to hold the Control key down until the Power Mac Debugger comes to the front.) This process causes the application to be automatically targeted for debugging. Code is **targeted** when the debugger is aware of it and has read its process information.

If you double-click an application's icon without holding down the Control key, the application launches normally; that is, it does not stop and is not targeted by the debugger. If you wish to target the application for debugging, follow one of the procedures in the next section, "Targeting and Stopping Your Code".

If you are debugging on a single machine, there are three other ways to do a break on launch:

- Use the Launch command from the Control menu. A dialog box appears, allowing you to choose an application to launch.
- Drag the application's icon onto the debugger's icon in the Finder.
- Use the Open command from the File menu, and select your application from the dialog box. In addition to launching your application, the debugger opens the symbol file associated with the application, if it can find it.

Targeting and Stopping Your Code

The previous section showed how to automatically target and stop your code by doing a break on launch. There are several other ways to target or stop your code:

- You can include `Debugger` and `DebugStr` calls in your program. Any time one of these is executed, the debugger is entered, the string (if any) is written to the Log window, and the process is automatically targeted.
- You can target an untargeted application (or a thread within an application, when using the application nub) by using the Process Browser. For information, see "The Process Browser" (page 73). Once your code is targeted, you can set breakpoints in order to stop it. Chapters 3 and 4 contain information on setting breakpoints.
- If you are using the serial nub, you can click the Stop icon (black square) from the control palette or choose Stop from the Control menu. Make sure that your application is the current process when you do this; for example, pull and hold down a menu in the target application while clicking Stop.

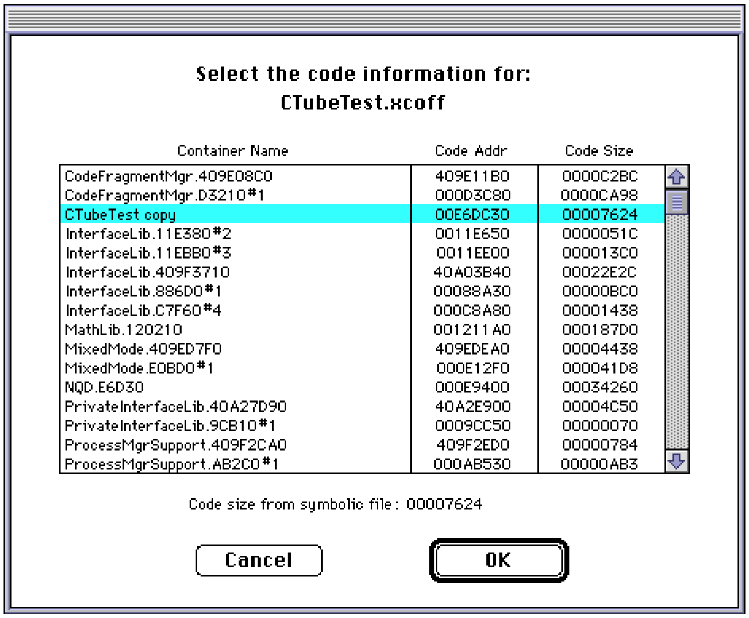
Mapping Symbols To Code

For you to debug at the source level, the debugger must associate the information in a symbol file with code running on the target machine. This process is called **mapping**. Any symbol file you open must be mapped to a code fragment. Normally, this mapping is automatic, and you do not need to do anything explicitly.

- The most common type of matching is done by name. If you have opened a symbol file called `MyApp.xcoff` or `MyApp.xSYM`, the debugger maps it to a loaded code fragment named MyApp.
- If there is no code fragment with a matching name, the debugger looks for a fragment having the same size as the size specified in the symbol file. If it finds one, the mapping takes place. If it finds more than one fragment with the same size, no mapping takes place.

  If you do not want mapping by size, you can uncheck the option "Always auto-map symbol file" in the General Preferences dialog box. For more information, see Appendix A.
- If the debugger cannot do the mapping or if you want to explicitly map the symbol file to a particular fragment, choose Map Symbol File from the Control menu. A dialog box (Figure 2-5) appears, allowing you to select a fragment to map to the symbol file.

Figure 2-5 Mapping the symbol file to a fragment



Select the code information for:
CTubeTest.xcoff

Container Name	Code Addr	Code Size
CodeFragmentMgr.409E08C0	409E11B0	0000C2BC
CodeFragmentMgr.D3210#1	000D3C80	0000CA98
CTubeTest copy	00E6DC30	00007624
InterfaceLib.11E380#2	0011E650	0000051C
InterfaceLib.11EBB0#3	0011EE00	000013C0
InterfaceLib.409F3710	40A03B40	00022E2C
InterfaceLib.886D0#1	00088A30	00000BC0
InterfaceLib.C7F60#4	000C8A80	00001438
MathLib.120210	001211A0	000187D0
MixedMode.409ED7F0	409EDEA0	00004438
MixedMode.E0BD0#1	000E12F0	000041D8
NQD.E6D30	000E9400	00034260
PrivateInterfaceLib.40A27D90	40A2E900	00004C50
PrivateInterfaceLib.9CB10#1	0009CC50	00000070
ProcessMgrSupport.409F2CA0	409F2ED0	00000784
ProcessMgrSupport.AB2C0#1	000AB530	00000AB3

Code size from symbolic file: 00007624

Note
The Browser window (or other code window) must be frontmost for the Map Symbol File command to be activated. Also, the fragments displayed in the Map Symbol File dialog box are those associated with the currently focused process. If the code you want to map is associated with a different process, you must change the current focus. See "The Current Focus" (page 76). ◆

C H A P T E R 3

The Debugger Windows

Contents

The Power Mac Debugger has several types of windows you can work with to control your program's execution and get information while debugging. This chapter describes what they look like and how you work with them.

Chapter 4, "Basic Debugging Tasks," goes into more detail about some of the tasks you can perform when using the debugger's windows.

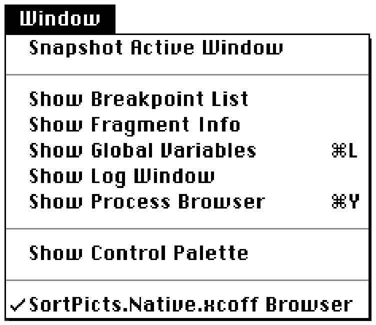
General Information About Debugger Windows

It's useful to think of the debugger's windows as falling into three major categories: windows that can have only one instance, those that can have any number of instances, and those that can have one instance per process.

Windows that can have only one instance are listed in the Window menu (Figure 3-1): Breakpoint List, Fragment Info, Global Variables, Log Window, Process Browser, and control palette.

You open them by choosing Show *windowname* from the Window menu. You can close them by choosing Close *windowname* from the Window menu.

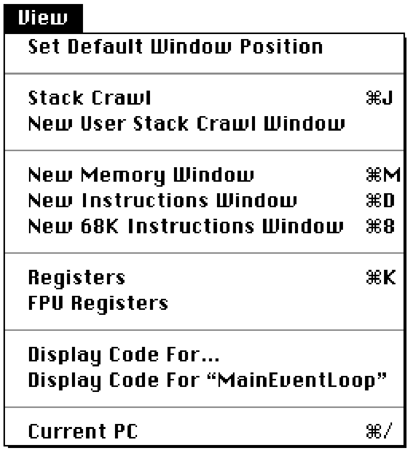
Figure 3-1 The Window menu



Windows that can have any number of instances are listed in the View menu (Figure 3-2): User Stack Crawl, Memory, Instructions, and 68K Instructions.

You open them by choosing New *windowname* Window from the View menu.

Figure 3-2 The View menu


View
Set Default Window Position
Stack Crawl ⌘J
New User Stack Crawl Window
New Memory Window ⌘M
New Instructions Window ⌘D
New 68K Instructions Window ⌘8
Registers ⌘K
FPU Registers
Display Code For…
Display Code For "MainEventLoop"
Current PC ⌘/

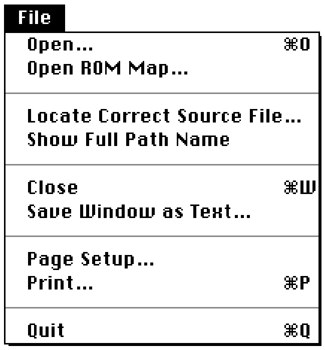
Windows that can have one instance per process, except for the Code Browser, are listed in the View menu: Stack Crawl, Registers, and FPU registers.

The Code Browser opens automatically whenever you open a symbol file (by choosing Open from the File menu). You open the other windows by choosing their name from the View menu.

When you open any window, its name appears at the bottom of the Window menu and you can choose it from there anytime to bring it to the front.

Any window can be closed by clicking its close box or by choosing Close from the File menu (Figure 3-3).

Figure 3-3 The File menu



Saving and Printing a Window’s Contents

To save a window’s contents, choose Save Window as Text from the File menu. The contents are saved in an MPW text file. To print the contents of any window, choose Print from the File menu.

For some windows, such as those displaying source code for a single function, the debugger saves or prints the entire scrollable contents of the window. For windows that display memory and instructions, and allow you to scroll over the entire address space, only the visible portion of the window is saved or printed.

Taking a Snapshot of a Window’s Contents

Information displayed in debugger windows is dynamic and can change every time a program stops. To preserve a window’s data at any point in time, choose Snapshot Active Window from the Window menu. The debugger creates a new window showing the visible contents of the original window. You cannot save this snapshot window. You can, however, print the window by choosing Print from the File menu.

Note
The contents of the snapshot window are fixed at the time the snapshot is taken. The window can be resized but can't, for example, be scrolled to reveal additional lines of code that were offscreen when the snapshot was taken. ◆

Remembering Window Positions

When you initially open a window, the debugger places it at a default location (near the upper-left corner of the screen). Often you will move the window to a convenient location as you are debugging.

For the windows listed in the Window menu (those that have only one instance), the debugger automatically remembers their positions and sizes. For example, if you close the Process Browser and then reopen it, it appears at its previous location.

For the windows listed in the View menu, which can have multiple instances, the debugger places each new instance of the window at a slight offset from where the previous one was opened. If you choose Set Default Window Position from the Window menu, the debugger saves the frontmost window's location and size so that the next time you open a window of that type, it appears in the remembered location; if you open more than one of the same type, the windows are staggered.

The Browser window is a special case. If you use debugger project files, the position and size of the Browser is remembered separately for each symbol file. See "Browser Preferences" (page 153) for more information about project files.

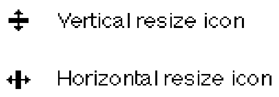
Multiple-Pane Windows

Some of the debugger windows (Browser, Process Browser, Stack Crawl, and Global Variables) have several panes. Figure 3-5 (page 50), for example, shows the three panes of the Browser window.

To save screen space, the titles of the panes (Files, Function, and Code in the Browser), by default, do not appear in the window. Figure 3-8 (page 54) shows the Browser as it appears without the window titles. To show the titles as they appear in Figure 3-5, check “Show pane titles” in General Preferences in the Edit menu.

To resize the panes, place the cursor over the split bar separating the panes; the cursor then changes to the resize icon (see Figure 3-4). Then click and drag away from the pane whose size you want to increase.

Figure 3-4 The resize icons



The Browser Window

The **Browser window** is a three-pane window displayed when you open a symbol file (by choosing Open from the File menu). You use this window to view the source code for selected functions in your target program. The Browser window is the only window that must be open in order to do source-level debugging. If you close it, the symbol file is no longer available.

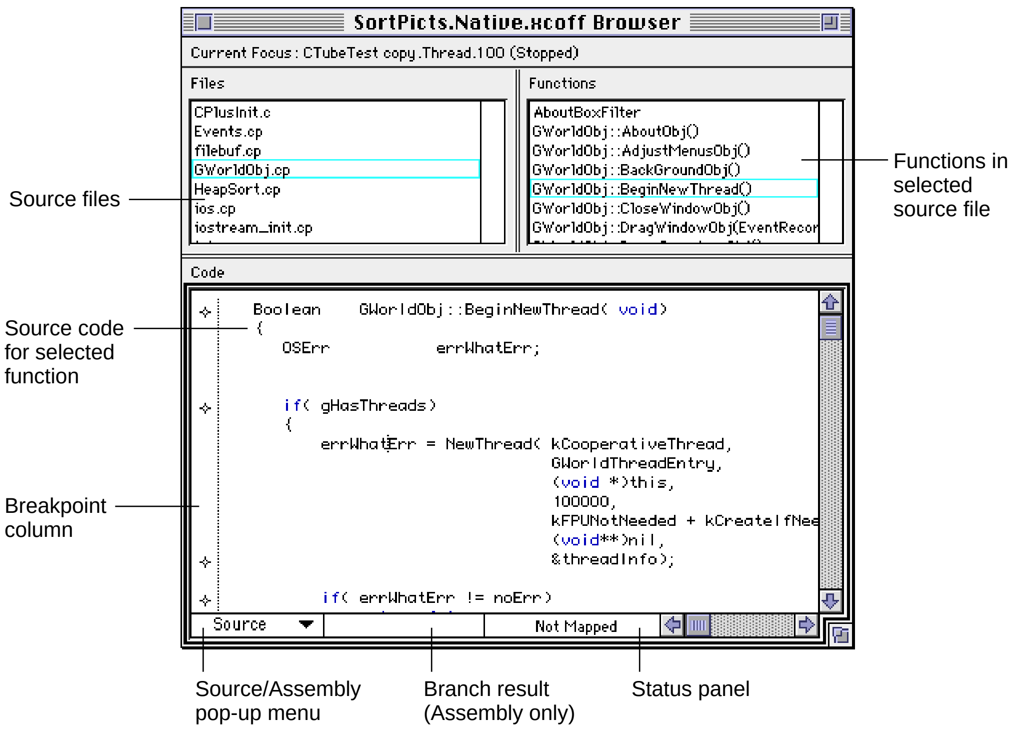
The Browser allows you to navigate easily through your source code, set breakpoints, and see the statements that are executing as you single-step.

You can open any number of symbol files, and the debugger creates a Browser window for each of them. The title of the window is the name of the symbol file, followed by the word Browser.

The Browser’s Panes

Figure 3-5 shows the Browser window and its parts.

Figure 3-5 The Browser window


Source files
Source code for selected function
Breakpoint column
Functions in selected source file
Source/Assembly pop-up menu
Branch result (Assembly only)
Status panel
Files
Functions
Code
Source
Not Mapped

SortPicts.Native.xcoff Browser

Current Focus : CTubeTest copy.Thread.100 (Stopped)

```
CPlusInit.c
Events.cp
filebuf.cp
GWorldObj.cp
HeapSort.cp
ios.cp
iostream_init.cp
```

```
AboutBoxFilter
GWorldObj::AboutObj()
GWorldObj::AdjustMenusObj()
GWorldObj::BackGroundObj()
GWorldObj::BeginNewThread()
GWorldObj::CloseWindowObj()
GWorldObj::DragWindowObj(EventRecor
```

```
Boolean    GWorldObj::BeginNewThread( void)
{
    OSErr           errWhatErr;


    if( gHasThreads)
    {
        errWhatErr = NewThread( kCooperativeThread,
                                GWorldThreadEntry,
                                (void *)this,
                                100000,
                                kFPUNotNeeded + kCreateIfNee
                                (void**)nil,
                                &threadInfo);

        if( errWhatErr != noErr)
```

The Browser window contains three panes, which display the following information:

- **Files.** The upper-left pane displays a list of source files, in alphabetical order. To select a source file, click its name. When this pane is active, you can type the first few letters of a file name to select it.
- **Functions.** The upper-right pane lists the functions in the selected source file, in alphabetical order. To select a function, click its name, or activate the Functions pane and type the first few letters of the function name.
- **Code.** The lower pane displays the code for the selected function. The pop-up menu at the lower-left corner of the window allows you to switch between source-code and assembly-language views of the selected function; the default is source code.

To the right of the pop-up menu is an area that shows the result of a branch instruction, when the Browser is displaying assembly code (see Figure 3-10).

The status panel at the bottom of the Browser tells you whether the code associated with the symbol file is currently mapped to targeted code, that is, whether it is ready to be debugged. See “Targeting and Stopping Your Code” and “Mapping Symbols To Code” (page 41) for information on how to target code and map it to a symbol file.

The currently active pane has a black border. You can navigate through the panes clockwise by pressing the Tab key or counterclockwise with Shift-Tab. You can also click whichever pane you want to make active. In Figure 3-5, the Functions pane is active and the currently selected function is highlighted. The Files pane shows which file is selected. The Code pane displays the source code for the selected function.

Sometimes the debugger cannot display the source code. For example, symbol information may not be available for that file (because the file was not compiled with the correct compiler option). At other times, the symbol information is available but the debugger cannot locate the source file. In this case, you are prompted with a standard file dialog box to locate the file.

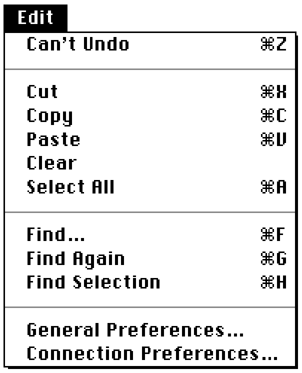
If you keep more than one version of source code on your disk, you may need to tell the debugger which version to use. To do this, choose Locate Correct Source File from the File menu. To find out which version the debugger is using, choose Show Full Path Name from the File menu.

If source code is not available for a function, the debugger attempts to display assembly code. Assembly code is available only when the application is targeted. If it is, the Code pane lists the assembly instructions along with their addresses in memory.

Working With the Code Pane

The Code pane displays functions formatted as they appear in the source file. By default, C and C++ code is color-coded: language keywords are shown in blue and comments are shown in red. You can turn color coding on and off by choosing General Preferences from the Edit menu (Figure 3-6) and clicking the “Use syntax coloring” option.

Figure 3-6 The Edit menu

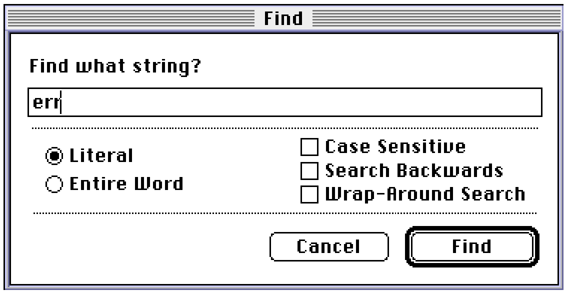


The text in the Code pane cannot be edited. However, you can select text in order to highlight certain statements or to copy and paste it. Here are some short cuts for doing so:

- To select a word, double-click it.
- To select an entire source-code statement, triple-click it. (If you then switch to the assembly-language view, all the instructions corresponding to the source statement are highlighted.)
- To select the text between delimiters, double-click one delimiter in that pair. The delimiters are parentheses (), brackets [], braces {}, angle brackets <>, single quotes ('), and double quotes (").
- To search for a string, Choose Find from the Edit menu (or press Command-F). A dialog box appears (Figure 3-7), containing several search options. In addition, you can use Find Again (Command-G) to search for the same text again, or Find Selection (Command-H) to search for the currently selected text. If you hold down the Shift key with these commands, the search proceeds backwards through the function.

These actions can also be performed in any other code window. See “Creating Other Code Windows” (page 54).

Figure 3-7 Finding text in a code window


Find

Find what string?

err

[✓ Literal] Entire Word

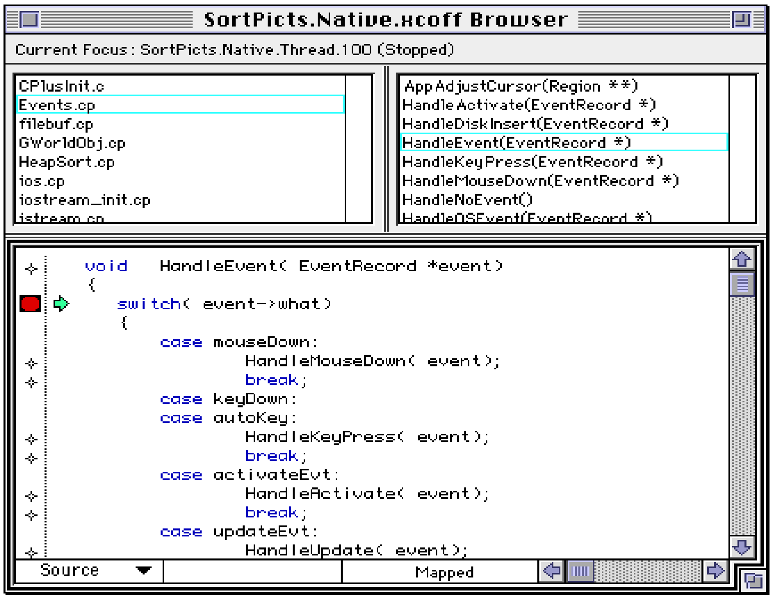
Case Sensitive

Search Backwards

Wrap-Around Search

In the Code pane, the area to the left of the vertical dotted line is the breakpoint column. To set a breakpoint, click one of the diamonds next to a source or assembly statement; a breakpoint icon appears in place of the diamond. Figure 3-8 shows an example of a breakpoint. See “Setting Breakpoints” (page 85) for information on the types of breakpoints you can set.

Figure 3-8 A breakpoint in the Code pane



SortPicts.Native.xcoff Browser

Current Focus : SortPicts.Native.Thread.100 (Stopped)

```
CPlusInit.c
Events.cp
filebuf.cp
GWorldObj.cp
HeapSort.cp
ios.cp
iostream_init.cp
istream.cp
```

```
AppAdjustCursor(Region **)
HandleActivate(EventRecord *)
HandleDiskInsert(EventRecord *)
HandleEvent(EventRecord *)
HandleKeyPress(EventRecord *)
HandleMouseDown(EventRecord *)
HandleNoEvent()
HandleOSEvent(EventRecord *)
```

```
void    HandleEvent( EventRecord *event)
{
    switch( event->what)
    {
        case mouseDown:
                HandleMouseDown( event);
                break;
        case keyDown:
        case autoKey:
                HandleKeyPress( event);
                break;
        case activateEvt:
                HandleActivate( event);
                break;
        case updateEvt:
                HandleUpdate( event);
```

The green arrow represents the **program counter (PC)**, which is the next statement or instruction to be executed. In Figure 3-8, the position of the PC shows that the breakpoint has just been hit. If you begin single-stepping, the program arrow moves; the breakpoint stays where it is. See “Controlling Program Execution” (page 96) for more information on single-stepping.

When your code is stopped at a breakpoint, or when you have single-stepped, the Code pane displays the function containing the current program counter. If you then select another function from the Functions pane, the PC is no longer visible; to get back to the function containing the PC, choose Current PC from the View menu.

Creating Other Code Windows

There may be occasions when you want to see more than one function at a time, or more than one view of a single function. You may, for example, want to compare source and assembly code for a given function. Or you may have narrowed a bug search to two or three routines and want them all on the screen simultaneously. The Browser can display only one routine, but you can “clone” its Code pane to create a new window by following these steps:

1. **Place the cursor in the Code pane of the Browser window.**
2. **Hold down the Option key. The cursor changes into a small window icon, as shown in Figure 3-9.**
3. **Click in the pane, then drag and release to create a code view window displaying the same function.**
4. **Select Source or Assembly from the pop-up menu in the newly created window to display the desired view.**

Figure 3-9 The clone-window cursor

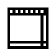
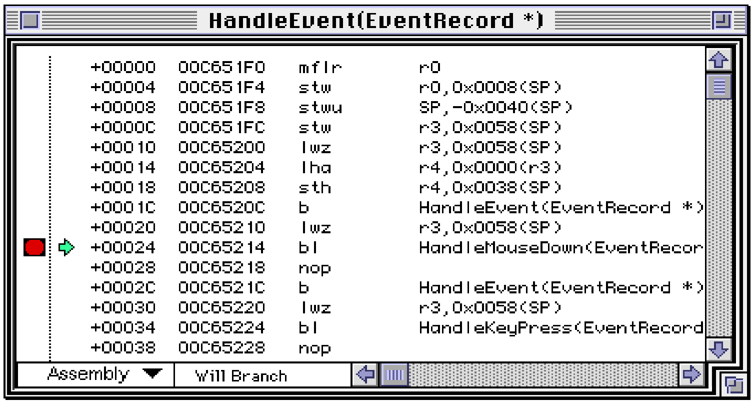


Figure 3-10 shows the assembly-language view of the code shown in Figure 3-8, displayed in its own code window. The breakpoint (and program counter) is at the assembly instruction corresponding with the source-code instruction in the other window.

Figure 3-10 A cloned window with assembly-code display

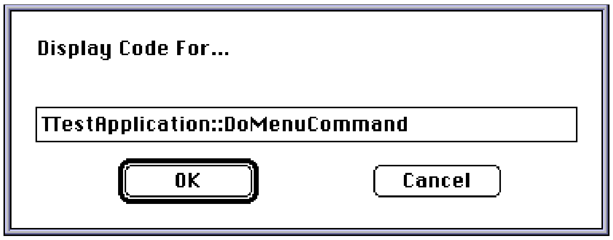


```
+00000  00C651F0  mflr   r0
+00004  00C651F4  stw    r0,0x0008(SP)
+00008  00C651F8  stwu   SP,-0x0040(SP)
+0000C  00C651FC  stw    r3,0x0058(SP)
+00010  00C65200  lwz    r3,0x0058(SP)
+00014  00C65204  lha    r4,0x0000(r3)
+00018  00C65208  sth    r4,0x0038(SP)
+0001C  00C6520C  b      HandleEvent(EventRecord *)
+00020  00C65210  lwz    r3,0x0058(SP)
+00024  00C65214  bl     HandleMouseDown(EventRecor
+00028  00C65218  nop
+0002C  00C6521C  b      HandleEvent(EventRecord *)
+00030  00C65220  lwz    r3,0x0058(SP)
+00034  00C65224  bl     HandleKeyPress(EventRecord
+00038  00C65228  nop
```

If you want to display a function in its own code window but don't remember which source file it is in, use one of the following commands from the View menu.

- **Display Code For.** Display Code For allows you to enter the name of a function in a dialog box, as shown in Figure 3-11. If the function exists, the debugger displays it in an independent code window. For C++ methods, you must give the full class name and function name, as shown here. You do not need to type parentheses or argument lists. (If there are two are more C++ functions with the same name but different argument lists, the debugger may not display the one you want.)

Figure 3-11 Locating a function's code



TTestApplication::DoMenuCommand

- **Display Code For Selection.** When the name of a function is selected in an active code window, you can use the Display Code For Selection command. The word “Selection” is replaced by the name of the function. For example, if you select the name `HandleEvent`, the menu command reads Display Code For “HandleEvent”. Choosing that command causes the code to be displayed in a new window.

Note
Remember to specify the complete name of a C++ method, for example, `MyClass::MyRoutine`. If you select the name `MyRoutine` inside a code window displaying another function of the same class and choose Display Code For MyRoutine, the routine will not be found. You must either type the complete name in the Display Code For dialog box or double-click the name in the Stack Crawl; see “The Stack Crawl Window” (page 63). ◆

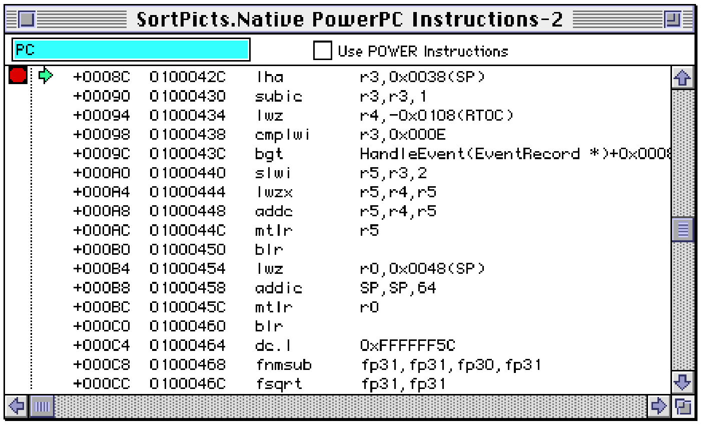
Instructions Windows

In addition to code windows displaying a single routine, the debugger provides instructions windows that can disassemble any area of memory. They are particularly useful for stepping through code that is not part of your application (for example, system code).

Disassembling PowerPC Code

An **instructions window** disassembles an area of memory to PowerPC instructions (Figure 3-12). To open one of these windows, choose New Instructions Window from the View menu (or press Command-D).

Figure 3-12 A PowerPC instructions window



SortPicts.Native PowerPC Instructions-2

PC ☐ Use POWER Instructions

```
+0008C  0100042C  lha      r3,0x0038(SP)
+00090  01000430  subic    r3,r3,1
+00094  01000434  lwz      r4,-0x0108(RTOC)
+00098  01000438  cmplwi   r3,0x000E
+0009C  0100043C  bgt      HandleEvent(EventRecord *)+0x000
+000A0  01000440  slwi     r5,r3,2
+000A4  01000444  lwzx     r5,r4,r5
+000A8  01000448  addc     r5,r4,r5
+000AC  0100044C  mtlr     r5
+000B0  01000450  blr
+000B4  01000454  lwz      r0,0x0048(SP)
+000B8  01000458  addic    SP,SP,64
+000BC  0100045C  mtlr     r0
+000C0  01000460  blr
+000C4  01000464  dc.l     0xFFFFFF5C
+000C8  01000468  fnmsub   fp31,fp31,fp30,fp31
+000CC  0100046C  fsqrt    fp31,fp31
```

The box at the upper-left corner of the window shows the starting address of the disassembly. If the debugger can convert the selection in the previous frontmost window to an address, it uses that address to begin the disassembly. Otherwise, it uses the program counter. To change the beginning address, enter a new value and press Return or Enter.

Unlike code windows, instructions windows are not restricted to a single function. In fact, you can scroll through the entire range of memory. To scroll up or down in the display, click the arrows; to move up or down one page, respectively, click above or below the thumb.

Note
In instructions windows, the thumb in the vertical scroll bar does not move. Because you can scroll through the entire range of memory, in a sense what you are viewing is always at the "center" of the display. ◆

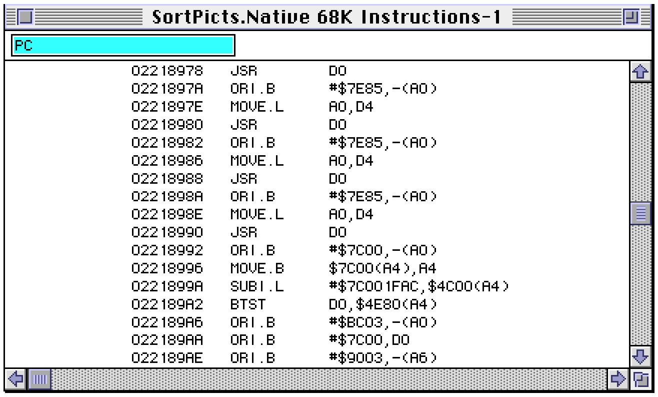
As with source-code displays, you can set breakpoints and single-step through code in instructions windows. You can set a breakpoint on any instruction by placing the cursor in the breakpoint column and clicking to the left of the desired instruction. See "Setting Breakpoints" (page 85) and "Controlling Program Execution" (page 96).

Note
Instruction displays use PowerPC mnemonics by default. The PowerPC instruction set is a subset of IBM's original POWER instruction set, which uses a different set of mnemonics. If you prefer to use the POWER mnemonics, click the checkbox in the window (Figure 3-12). You can also change the default setting by using General Preferences in the Edit menu. See Appendix A, "Debugger Preferences," for more information. ◆

Disassembling 680x0 Code

A **68K instructions window** disassembles an area of memory to 680x0 instructions (Figure 3-13). To open one of these windows, choose New 68K Instructions Window from the View menu.

Figure 3-13 A 68K instructions window



```
02218978   JSR       D0
0221897A   ORI.B     #$7E85,-(A0)
0221897E   MOVE.L    A0,D4
02218980   JSR       D0
02218982   ORI.B     #$7E85,-(A0)
02218986   MOVE.L    A0,D4
02218988   JSR       D0
0221898A   ORI.B     #$7E85,-(A0)
0221898E   MOVE.L    A0,D4
02218990   JSR       D0
02218992   ORI.B     #$7C00,-(A0)
02218996   MOVE.B    $7C00(A4),A4
0221899A   SUBI.L    #$7C001FAC,$4C00(A4)
022189A2   BTST      D0,$4E80(A4)
022189A6   ORI.B     #$BC03,-(A0)
022189AA   ORI.B     #$7C00,D0
022189AE   ORI.B     #$9003,-(A6)
```

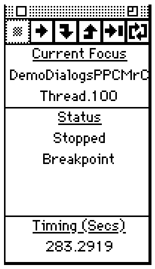
The box at the upper-left corner of the window shows the starting address of the disassembly. If the debugger can convert the selection in the previous frontmost window to an address, it uses that address to begin the disassembly. Otherwise, it uses the program counter. To change the beginning address, enter a new value and press Return or Enter.

Note
The Power Mac Debugger does not allow you to set breakpoints in or step through 680x0 code. For that reason, the 68K Instructions window does not show the current PC and does not have a breakpoint column. The validity of the disassembled instructions depends on your specifying an address that actually contains 680x0 code. ◆

The Control Palette

When you launch the debugger, the **control palette** appears (Figure 3-14). It is a floating palette that appears in front of all other open windows, showing information about your program's status and containing a row of icons that you can click to control the execution of your program.

Figure 3-14 The control palette



Current Focus
DemoDialogsPPCMrC
Thread.100
Status
Stopped
Breakpoint
Timing (Secs)
283.2919

To shrink the control palette so that it shows only the control icons, click the zoom box in the upper-right corner. To close it, click the close box or choose Close Control Palette from the Window menu. To reopen it, choose Show Control Palette from the Window menu.

Program Status Information

The control palette displays the following status information:

- **Current Focus**. At any given time, a number of applications may be targeted by the debugger. Only one of these (at most) has the **current focus.** All the commands in the control palette and Control menu (such as the Step Over command) operate on this process. To single-step in a different process, you must change the current focus. See “The Process Browser” (page 73) for more information on setting the current focus and on targeting and untargeting processes.
- **Status**. A targeted application can be in one of several states:
  - **Running**.The application is executing. It has an entry in the Process menu and the user can interact with it (assuming it has a user interface).
  - **Stopped.** The application has been stopped under debugger control. Its entry in the system Process menu is dimmed. The control palette displays the reason for the stop: Breakpoint, Stepped, DebugStr, or User Break.
  - **Suspended.** A thread of the application has been temporarily halted and may be continued later. This state applies only to cooperative threads under the Thread Manager.
- **Timing.** If you set a breakpoint, run a process, and reach the breakpoint, the debugger displays the time elapsed on the target machine between the start of execution and the break.

In the control palette shown in Figure 3-14, the application DemoDialogsPPCMrC has the current focus, and it is stopped at a breakpoint. The name “Thread.100” refers to the application’s main thread of execution. For more information on threads, see “The Process Browser” (page 73).

Program Control Commands

The icons in the control palette (Figure 3-15) represent the most common commands to control the execution of a program. Table 3-1 shows the action that takes place when you click each icon and their Control menu equivalents.

Figure 3-15 Control palette icons

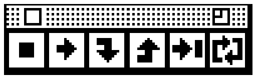


Table 3-1 Control palette icons and their Control menu equivalents

Palette icon	Control menu equivalent	Effect
■	Stop	Stops the application (available only when using the serial nub).
➡	Run (Command-R)	Resumes execution of the application. If already running, the application comes to the front.
⬇	Step Into (Command-T)	Executes the next statement. If it is a function call, stepping continues with the first statement of the called function. The new routine is displayed in the Browser window.
⬆	Step Out (Command-U)	Completes execution of the current routine. Stepping continues following the statement that called the current routine.
➡▮	Step Over (Command-S)	Executes the next statement. If it is a function call, the function is executed in its entirety; stepping continues with the first statement following the function call.
↻	Turn Continuous Step On / Off	When you click this icon, it highlights. When you then choose Step, Step Into, or Step Out, the debugger repeatedly executes that command until a breakpoint is reached. To stop the continuous stepping, click the icon again.

Note
All commands operate on the application that has the current focus. ◆

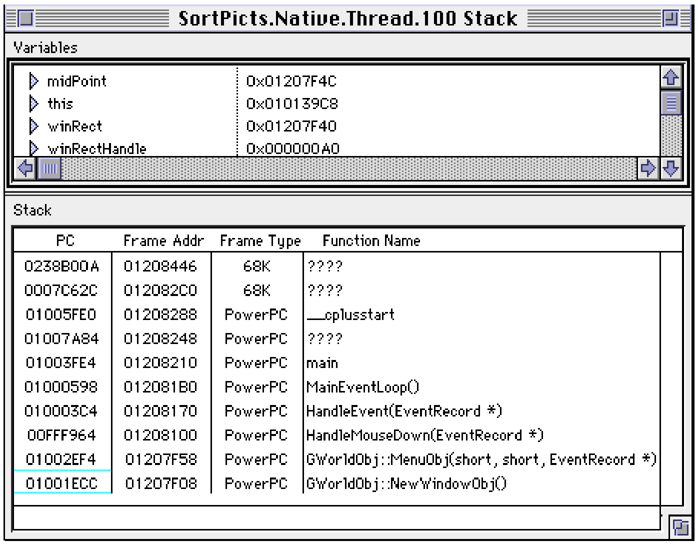
When stepping through code, if the active window is displaying a source view, the program steps one source statement at a time. If the active window is displaying an assembly-language view, the program steps one machine instruction at a time. If the active window is not a code window, the debugger uses the last code view that had been displayed to determine whether to step in source or assembly.

For additional information on these commands, as well as other commands in the Control menu, see "Controlling Program Execution" (page 96).

The Stack Crawl Window

The **Stack Crawl window** (Figure 3-16) is a two-pane window that displays information about routines on the stack. To open this window, choose Stack Crawl from the View menu (or press Command-J).

Figure 3-16 The Stack Crawl window



midPoint	0x01207F4C
this	0x010139C8
winRect	0x01207F40
winRectHandle	0x000000A0

PC	Frame Addr	Frame Type	Function Name
0238B00A	01208446	68K	????
0007C62C	012082C0	68K	????
01005FE0	01208288	PowerPC	__cplusstart
01007A84	01208248	PowerPC	????
01003FE4	01208210	PowerPC	main
01000598	012081B0	PowerPC	MainEventLoop()
010003C4	01208170	PowerPC	HandleEvent(EventRecord *)
00FFF964	01208100	PowerPC	HandleMouseDown(EventRecord *)
01002EF4	01207F58	PowerPC	GWorldObj::MenuObj(short, short, EventRecord *)
01001ECC	01207F08	PowerPC	GWorldObj::NewWindowObj()

The lower pane shows the call chain of routines on the stack. The upper pane shows the variables corresponding to each routine. The active pane is indicated by a black border (the lower pane in Figure 3-16). You can switch between the panes by using the Tab key or by clicking in a pane.

You can resize the panes by placing the cursor over the split lines to display the vertical resize icon, then clicking and dragging. You can also resize the columns in the upper pane.

Navigating the Call Chain

In Figure 3-16, the current routine (that is, the routine containing the current program counter) is `GWorldObj::NewWindowObj`. It is displayed at the bottom of the Stack pane. The routine that called it, `GWorldObj::MenuObj`, is above it in the display. That routine, in turn, was called by the routine in the row above it. This list of routines represents the **call chain**.

This display can be very useful for understanding how certain problems occurred. When you enter the debugger (for instance, with a `DebugStr` call), the Code Browser shows the current routine. Seeing only that routine may not help you to pinpoint the problem, particularly if it is a routine that is called from several places. The Stack Crawl provides more information by giving you a view of the path your program took to reach the current routine.

Each row in the display shows the following information:

- **PC**. For the current routine, this value represents the current program counter. For other routines in the call chain, this value represents the address of the instruction that called the next routine.

  If you double-click an address in the “PC” column, an instructions window appears, showing assembly code starting at that address (PowerPC or 680x0, as appropriate). For more information, see “Instructions Windows” (page 57).

- **Frame address**. The address of the **stack frame**, which is an area on the stack containing data related to the routine. The current routine’s stack frame address is equal to the current stack pointer (SP), that is, the top of the stack. Since the PowerPC stack grows toward low memory, the frame address of each calling routine is higher in memory.

  If you double-click a frame address, the debugger displays a memory window, with the memory display beginning at the frame address. For more information, see “Memory Windows” (page 67).

- **Frame type**. The routine’s instruction set, that is, either PowerPC or 680x0.

- **Function name**. The routine's name (if it is known). If the name is not known, it is shown as four question marks (????).

  If you double-click the name, the routine is displayed in the Browser window. To display the routine in a separate code window, hold the Option key down while double-clicking. If the routine's name is "????", the debugger displays an instructions window starting at the address listed in the "PC" column.

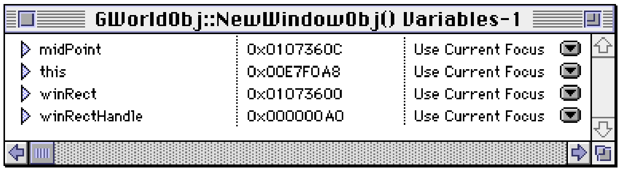
Examining Local Variables

The upper pane of the Stack Crawl window lists the variables corresponding to the stack frame selected in the lower pane. This list includes the routine's local variables as well the parameters that were passed to it. To select a routine, click anywhere in its row in the lower pane; the variables are then displayed in the upper pane. The variables' names appear to the left of the dotted line, and their values appear to the right.

Some variables have disclosure triangles to the left of their names. The triangles indicate variables that are structures, arrays, pointers or handles. If you click a triangle, it points down and shows the contents of the variable in expanded form. You can also display the variable in its own window by double-clicking its name. See "Displaying and Editing Variables" (page 101) for more information on interpreting variable displays.

You can create a separate copy, or "clone", of the variables pane. To do this, hold down the Option key, click in the pane and drag. The pane then appears in a new window, as shown in Figure 3-17. Cloning allows you to maintain a view of a given routine's variables regardless of what the Stack Crawl window shows. The variables' values are automatically updated each time the program stops.

Figure 3-17 A variable window cloned from the stack variables pane



```
GWorldObj::NewWindowObj() Variables-1
midPoint        0x0107360C   Use Current Focus
this            0x00E7F0A8   Use Current Focus
winRect         0x01073600   Use Current Focus
winRectHandle   0x000000A0   Use Current Focus
```

Note
The third column of this clone window shows the context in which the variable is being evaluated. By default, this context is the current focus. For more information on setting the context of a variable, see "Global Variables" (page 101).

Displaying a User Stack Crawl

A **User Stack Crawl** window displays a stack starting at an arbitrary address. To open one of these windows, choose New User Stack Crawl from the View menu. By default, the stack displays a copy of the existing stack crawl in a new window. At the top of the window is a box called "Frame Address", in which you can type any address. If that address is a valid stack frame, the rest of the stack is displayed.

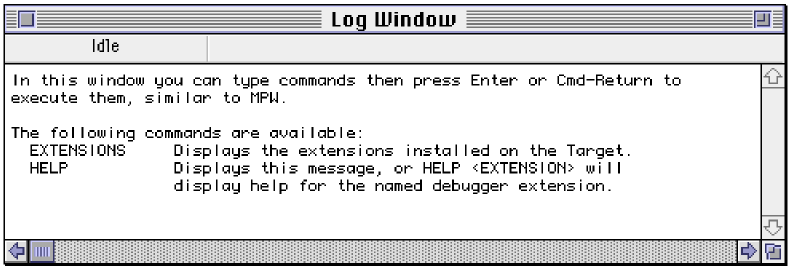
The Log Window

The **Log window** is a text window that

- allows you to execute commands (in particular, debugger extensions)
- displays the output of `DebugStr` calls and debugger extensions

To open this window, choose Show Log from the Window menu. Figure 3-18 shows the Log window as it first appears.

Figure 3-18 The Log window



```
In this window you can type commands then press Enter or Cmd-Return to
execute them, similar to MPW.

The following commands are available:
  EXTENSIONS     Displays the extensions installed on the Target.
  HELP           Displays this message, or HELP <EXTENSION> will
                 display help for the named debugger extension.
```

As in the MPW Shell, you can execute commands by typing them and then pressing the Enter key. Entering "help" gives you a list of available commands.

You can execute debugger extensions from this window. See Appendix C, "Debugger Extensions," for information on creating and executing debugger extensions.

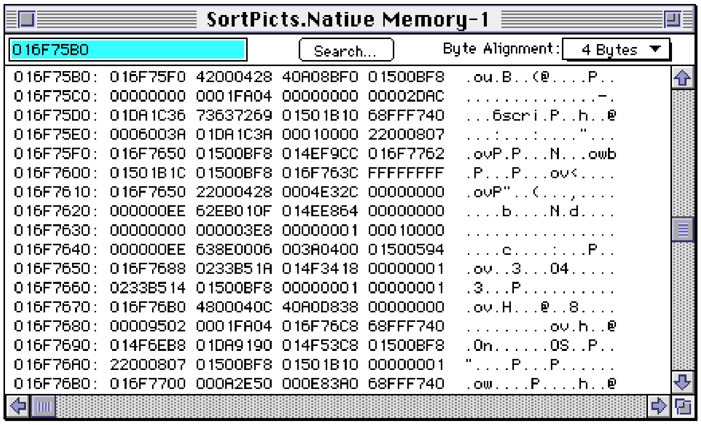
Whenever a `DebugStr` call executes, The Log window opens and displays the string that was passed as a parameter to `DebugStr`. If you want the debugger to display the date and time next to Log window output, choose General Preferences from the Edit menu and check the "Time stamp log entries" option.

You can save the contents of the Log window to an MPW text file by choosing Save Window as Text from the File menu.

Memory Windows

Memory windows display the contents of memory starting at a given address (Figure 3-19). To open a memory window, choose New Memory Window from the View menu (or press Command-M).

Figure 3-19 A memory window



```
016F75B0
016F75B0:  016F75F0 42000428 40A08BF0 01500BF8  .ou.B..(@....P..
016F75C0:  00000000 0001FA04 00000000 00002DAC  ..............-.
016F75D0:  01DA1C36 73637269 01501B10 68FFF740  ...6scri.P..h..@
016F75E0:  0006003A 01DA1C3A 00010000 22000807  ...:...:....."...
016F75F0:  016F7650 01500BF8 014EF9CC 016F7762  .ovP.P...N...owb
016F7600:  01501B1C 01500BF8 016F763C FFFFFFFF  .P...P...ov<....
016F7610:  016F7650 22000428 0004E32C 00000000  .ovP"..(...,....
016F7620:  000000EE 62EB010F 014EE864 00000000  ....b....N.d....
016F7630:  00000000 000003E8 00000001 00010000  ................
016F7640:  000000EE 638E0006 003A0400 01500594  ....c....:...P..
016F7650:  016F7688 0233B51A 014F3418 00000001  .ov..3...04.....
016F7660:  0233B514 01500BF8 00000001 00000001  .3...P..........
016F7670:  016F76B0 4800040C 40A0D838 00000000  .ov.H...@..8....
016F7680:  00009502 0001FA04 016F76C8 68FFF740  .........ov.h..@
016F7690:  014F6EB8 01DA9190 014F53C8 01500BF8  .On......OS..P..
016F76A0:  22000807 01500BF8 01501B10 00000001  "....P...P......
016F76B0:  016F7700 000A2E50 000E83A0 68FFF740  .ow....P....h..@
```

Using memory windows, you can view and edit any area of memory. You can also search for any value in memory.

The box at the upper-left corner of the window shows the starting address of the display. When you choose New Memory Window, if the debugger can convert the selection in the frontmost window to an address, it uses that address to begin the display. Otherwise, it uses the stack pointer. To change the address where the display begins, enter a new value (an address, a register, or an arithmetic expression) in the box and press Return or Enter.

Note
When a memory window is active, pressing Shift-Command-M causes the display to scroll to whatever address is selected in the window. ◆

On the Mac OS platform, addressable memory ranges from $0000 0000 to $FFFF FFFF, a range of 4 gigabytes (GB). You can use the scroll bar to move the display up and down through the entire range of memory. Click above or below the thumb to move up or down one page. If you attempt to show an area of memory that is not valid, the debugger displays an alert.

Note
Unlike the thumb in most scrolling windows, the thumb in the vertical scroll bar does not move. The reason is that you can scroll through the entire range of memory, thus in a sense what you are viewing is always the "center" of the display. ◆

Each row of the memory display shows 16 bytes of data in hexadecimal and ASCII format. By default, the hexadecimal display shows the data in groups of 4 bytes (8 hexadecimal digits). To change the byte grouping, select "2 Bytes" or "1 Byte" from the Byte Alignment pop-up menu. To change the byte grouping default, choose General Preferences from the Edit menu; the "Other" category has a "Default mem view byte grouping" item with a pop-up menu for setting the value.

Editing Memory

You can edit any area of memory using a memory display window. Here are the steps:

1. **Click the value you want to change.**

   You can select groupings of 4 bytes, 2 bytes, or 1 byte, depending on the grouping preference you have set. You can select the next group by pressing the Tab key, and the previous group by pressing Shift-Tab.

2. **Enter the new value.**

   You are allowed to enter no more than the number of bytes selected. For example, if the byte grouping is 4, you can enter up to eight hexadecimal digits. If you enter fewer digits, the value is shown right justified, padded with zeros on the left.

3. **Press the Return, Enter, or Tab key to make the change.**

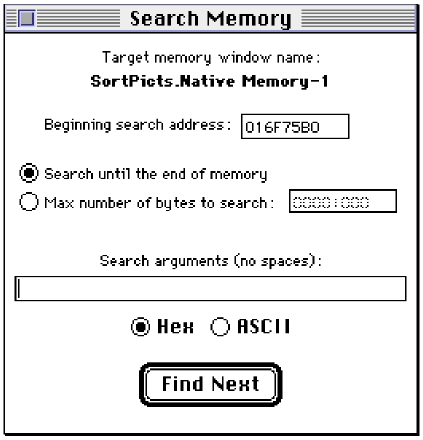
4. **To undo a change, choose Undo from the Edit menu.**

▲ **WARNING**
Changing memory can be dangerous unless you are absolutely sure what you are doing. The debugger cannot protect you from a crash if you accidentally change data that is important to an application or the system. ▲

Searching Memory

When a memory window is frontmost, you can search for any string or other value in memory. To do this, click the Search button (see Figure 3-19). A Search Memory window like the one in Figure 3-20 appears.

Figure 3-20 The Search Memory window


Search Memory
Target memory window name:
SortPicts.Native Memory-1
Beginning search address: 016F75B0
[✓ Search until the end of memory]
Max number of bytes to search: 0000:000
Search arguments (no spaces):
[✓ Hex] ASCII
Find Next

This window is associated with the foreground memory window; when you close the memory window, the Search Memory window also closes. It is possible to have several Search Memory windows open, each associated with a memory window.

When searching memory, you can specify

- an ASCII or hexadecimal value to search for
- a location in memory to begin the search

  By default, this is the value displayed in the memory window' upper-left box.
- whether you want to search through all of memory or only a specific range of addresses

You can stop a search at any time by pressing Command-period.

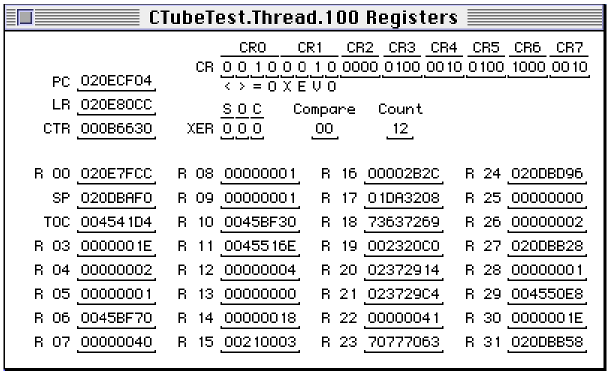
The Register Windows

The debugger provides two windows that allow you to display and change the values of PowerPC registers. One displays general-purpose registers, and the other displays floating-point registers.

General-Purpose Registers

The **Registers window** (Figure 3-21) displays the values of the PowerPC general-purpose registers. To open this window, choose Registers from the View menu (or press Command-K).

Figure 3-21 The general-purpose registers



```
CTubeTest.Thread.100 Registers

                          CR0     CR1    CR2  CR3  CR4  CR5  CR6  CR7
                      CR  0 0 1 0 0 0 1 0 0000 0100 0010 0100 1000 0010
  PC  020ECF04            < > = 0 X E V 0
  LR  020E80CC            S O C   Compare  Count
 CTR  000B6630        XER 0 0 0     00      12

R 00  020E7FCC   R 08  00000001   R 16  00002B2C   R 24  020DBD96
  SP  020DBAF0   R 09  00000001   R 17  01DA3208   R 25  00000000
 TOC  004541D4   R 10  0045BF30   R 18  73637269   R 26  00000002
R 03  0000001E   R 11  0045516E   R 19  002320C0   R 27  020DBB28
R 04  00000002   R 12  00000004   R 20  02372914   R 28  00000001
R 05  00000001   R 13  00000000   R 21  023729C4   R 29  004550E8
R 06  0045BF70   R 14  00000018   R 22  00000041   R 30  0000001E
R 07  00000040   R 15  00210003   R 23  70777063   R 31  020DBB58
```

The Registers window displays the 32 general-purpose registers, the branch unit registers, and the program counter.

- The general-purpose registers, the program counter (PC), the Link Register (LR), and the Count Register (CTR) are displayed in hexadecimal form. You can change the value of a register by clicking its value, entering a new value, and pressing Enter or Tab. To undo a change, choose Undo from the Edit menu.

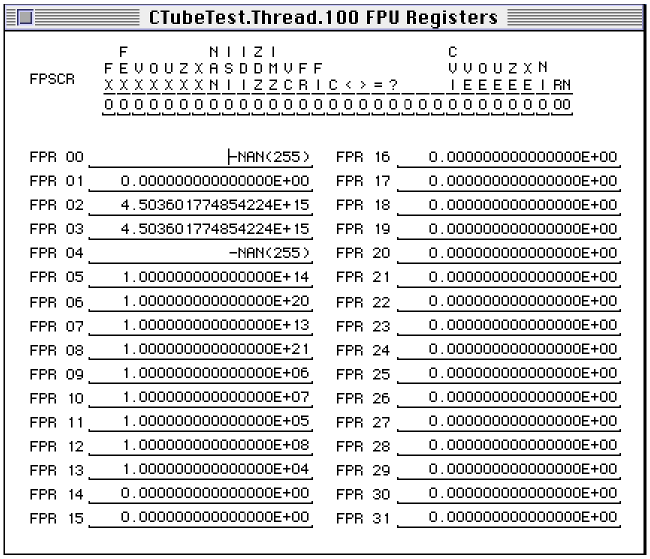
- The Condition Register (CR) and the Summary Overflow (S), Overflow (O), and Carry (C) bits of the Fixed-Point Exception Register (XER) are displayed in binary. Click a bit to toggle its value.

Note
By convention, R1 is the stack pointer and R2 points to the Table of Contents (TOC) for the currently executing fragment. R3 is used by C functions to store small result values (short, long, pointer). ◆

Floating-Point Registers

The **FPU Registers window** displays the values of the PowerPC floating-point registers (Figure 3-22). To open this window, choose FPU Registers from the View menu.

Figure 3-22 The floating-point registers



```
CTubeTest.Thread.100 FPU Registers

         F         N I I Z I                        C
         F E V O U Z X A S D D M V F F               V V O U Z X N
FPSCR    X X X X X X X N I I Z Z C R I C < > = ?     I E E E E E I RN
         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 00

FPR 00              -NAN(255)   FPR 16  0.000000000000000E+00
FPR 01  0.000000000000000E+00   FPR 17  0.000000000000000E+00
FPR 02  4.503601774854224E+15   FPR 18  0.000000000000000E+00
FPR 03  4.503601774854224E+15   FPR 19  0.000000000000000E+00
FPR 04              -NAN(255)   FPR 20  0.000000000000000E+00
FPR 05  1.000000000000000E+14   FPR 21  0.000000000000000E+00
FPR 06  1.000000000000000E+20   FPR 22  0.000000000000000E+00
FPR 07  1.000000000000000E+13   FPR 23  0.000000000000000E+00
FPR 08  1.000000000000000E+21   FPR 24  0.000000000000000E+00
FPR 09  1.000000000000000E+06   FPR 25  0.000000000000000E+00
FPR 10  1.000000000000000E+07   FPR 26  0.000000000000000E+00
FPR 11  1.000000000000000E+05   FPR 27  0.000000000000000E+00
FPR 12  1.000000000000000E+08   FPR 28  0.000000000000000E+00
FPR 13  1.000000000000000E+04   FPR 29  0.000000000000000E+00
FPR 14  0.000000000000000E+00   FPR 30  0.000000000000000E+00
FPR 15  0.000000000000000E+00   FPR 31  0.000000000000000E+00
```

This window shows the 32 floating-point registers and the Floating-Point Status and Control Register (FPSCR).

- By default, the floating-point register values are displayed in scientific (float) notation. You can change the display of an individual register by selecting the register and choosing either View as Float or View as Hexadecimal from the Evaluate menu.

  To change the value of a register, select the register, enter the new value, and press Enter or Tab. To undo a change, choose Undo from the Edit menu.
- The FPSCR value is displayed in binary form. Click a bit to toggle its value.

The Process Browser

The **Process Browser window** (Figure 3-23) displays a list of all currently executing native PowerPC processes and threads on the target machine. To open this window, choose Show Process Browser from the Window menu (or press Command-Y).

The Process Browser allows you to

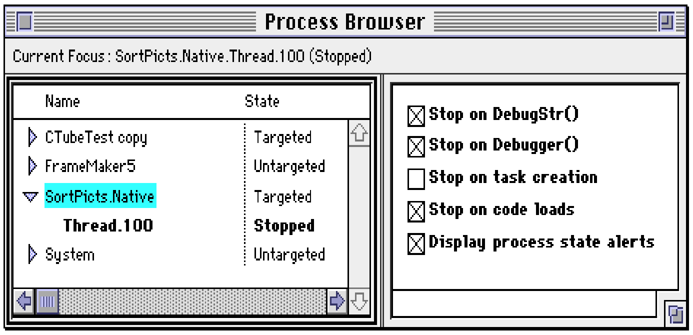
- see which processes and threads are executing on the target machine
- target and untarget specific processes and threads for debugging
- select which process or thread you want to be the focus of program control commands (for instance, single-stepping)
- set preferences for when the debugger should be entered, on a process basis

A **process** is equivalent to a running application on the Mac OS.

With System 7.5 and later, a programmer can use the Thread Manager to create multiple cooperative **threads** of execution within a single process. The Power Mac Debugger provides support for debugging separate threads within a given process, if you use the application nub (Power Mac DebugServices). For example, you can set a breakpoint that causes the program to stop at a particular line of code, but only if a specific thread is executing. The Process Browser displays information for all threads on the system.

The following sections describe the information displayed in the Process Browser.

Figure 3-23 The Process Browser



Current Focus : SortPicts.Native.Thread.100 (Stopped)

Name	State
CTubeTest copy	Targeted
FrameMaker5	Untargeted
SortPicts.Native	Targeted
Thread.100	**Stopped**
System	Untargeted

- [✓] **Stop on DebugStr()**
- [✓] **Stop on Debugger()**
- [ ] **Stop on task creation**
- [✓] **Stop on code loads**
- [✓] **Display process state alerts**

The Process List

The left pane of the Process Browser displays a list of the processes running on the target machine. This list includes all native PowerPC applications, as well as the System process. (Emulated 680x0 applications are not listed.)

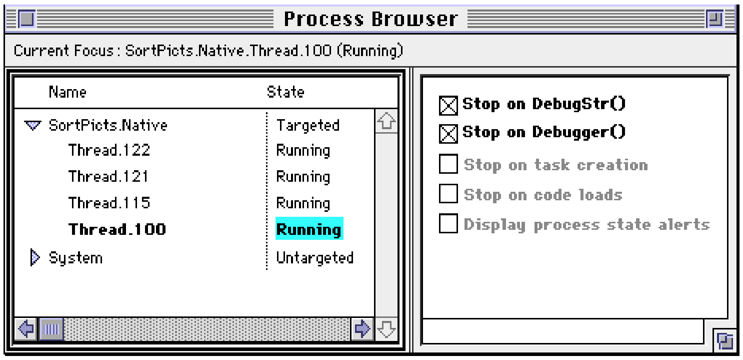
To the right of each process name is displayed its state—either targeted or untargeted. If a process is untargeted, it is not ready to be debugged. A process is untargeted by default unless you have launched it under debugger control or used one of the other methods mentioned in Chapter 2.

To target an untargeted process, you can

- double-click its name in the Process Browser
- select its name and choose Target from the Control menu
- generate an exception in the target application (for example, with a `DebugStr` call)

When a process is targeted, the Process Browser displays additional information. In Figure 3-23, note the disclosure triangles to the left of the process names. When you click a triangle, it points downward to reveal one or more items (if the process is targeted). These items represent the threads associated with a given process. Typically, each targeted process has a single thread associated with it, known as the main thread. An application’s main thread is always named Thread.100, by convention. Here, the SortPicts.Native application has a single main thread of execution. If new threads are created using the Thread Manager, they show up in the Process Browser and can be individually targeted if desired, as shown in Figure 3-24.

Figure 3-24 A multithreaded application



Process Browser

Current Focus : SortPicts.Native.Thread.100 (Running)

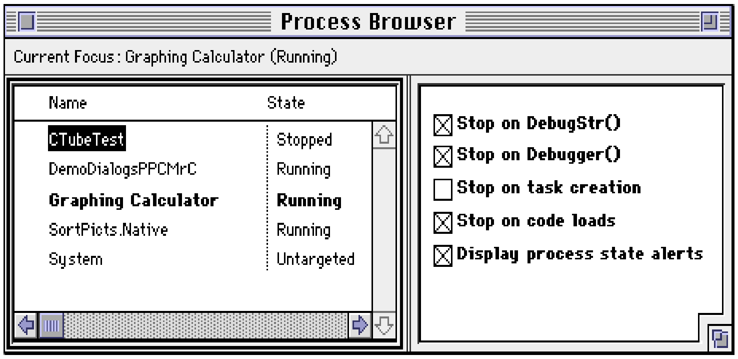
Name	State
SortPicts.Native	Targeted
Thread.122	Running
Thread.121	Running
Thread.115	Running
Thread.100	**Running**
System	Untargeted

- [x] **Stop on DebugStr()**
- [x] **Stop on Debugger()**
- [ ] Stop on task creation
- [ ] Stop on code loads
- [ ] Display process state alerts

In this example, each thread has a state displayed to the right of its name. A thread can be running, stopped or suspended. See “Program Status Information” (page 60) for descriptions of these states.

If you are using the serial nub (PPC Debugger Nub), you cannot debug individual threads. Therefore, the Process Browser displays processes only. See Figure 3-25, which lists five processes but shows no individual threads.

Figure 3-25 The Process Browser when using the serial nub



Process Browser

Current Focus : Graphing Calculator (Running)

Name	State
CTubeTest	Stopped
DemoDialogsPPCMrC	Running
Graphing Calculator	**Running**
SortPicts.Native	Running
System	Untargeted

- [✓] Stop on DebugStr()
- [✓] Stop on Debugger()
- [ ] Stop on task creation
- [✓] Stop on code loads
- [✓] Display process state alerts

To untarget a process, select its name in the Process Browser and choose Untarget from the Control menu.

The Current Focus

Note that in Figure 3-24 the entry for SortPicts.Native.Thread.100 is in boldface. This shows that it has the current focus—that all the commands in the control palette and Control menu apply to this process. A Step Over command, for example, causes the currently focused process to single-step, whether or not other processes are targeted. If you wish to single-step in a different process, you must change the current focus.

Note
When using the application nub, the current focus technically refers to a specific thread. If, however, the process has only a single thread, the focus can be considered to belong to the process. ◆

The current focus is also displayed at the top of the window, as well as in the control palette. To change the current focus, you can double-click the name of a process *or* one of its threads.

- If you double-click the name of an untargeted process, the process's state changes to targeted and its main thread (Thread.100) appears in boldface.

- If you double-click the name of a targeted thread, the thread's name changes to boldface and it becomes the current focus.

The current focus is automatically changed when

- you use the Launch command to launch an application
- an application hits a breakpoint or other exception (such as a `DebugStr` call)

Application Preferences

The right pane of the Process Browser has a series of checkboxes, which specify how a process can stop and enter the debugger, as shown in Figure 3-23. These items can be changed individually for any process. By default, some of the items are checked and some are unchecked. You can change the defaults, if you wish, by choosing General Preferences from the Edit menu and selecting items in the Process Control section. See Appendix A, "Debugger Preferences," for more information on the preferences you can set.

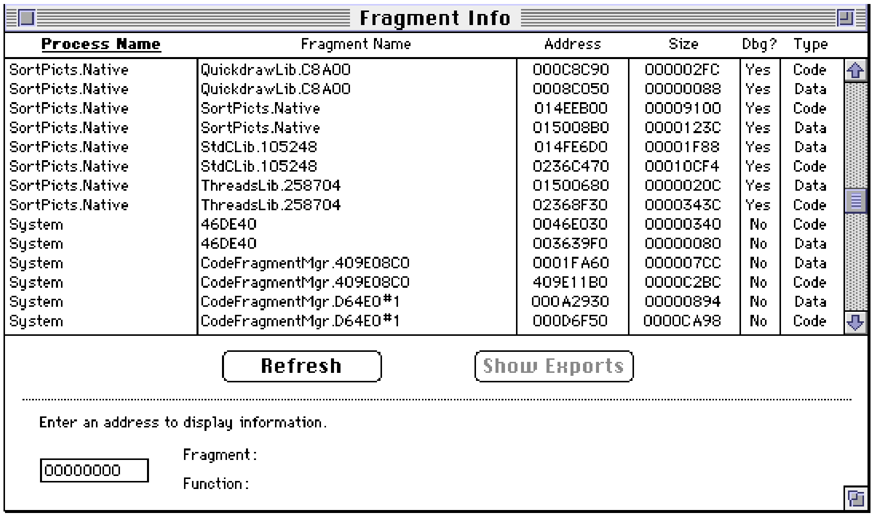
Here are the items that you can check:

- **Stop on DebugStr()** and **Stop on Debugger().** The debugger is entered whenever this process executes one of these calls. These items can also be checked or unchecked for individual threads.
- **Stop on code loads.** The debugger is entered any time a code fragment gets loaded. Stopping on code loads is useful if you want to debug some code, such as a stand-alone code resource, that your application loads. This option does not cause a newly launched process to stop when its code fragment is loaded; for that, you must explicitly break on launch.
- **Display process state alerts.** The debugger displays alerts when a process is launched under debugger control or when a process quits.
- **Stop on task creation.** The debugger stops whenever an application creates a new thread.

The Fragment Info Window

The **Fragment Info window** (Figure 3-26) allows you to get information about all code fragments on the target machines (not just those being debugged). To open this window, choose Show Fragment Info from the Window menu.

Figure 3-26 The Fragment Info window



Process Name	Fragment Name	Address	Size	Dbg?	Type
SortPicts.Native	QuickdrawLib.C8A00	000C8C90	000002FC	Yes	Code
SortPicts.Native	QuickdrawLib.C8A00	0008C050	00000088	Yes	Data
SortPicts.Native	SortPicts.Native	014EEB00	00009100	Yes	Code
SortPicts.Native	SortPicts.Native	015008B0	0000123C	Yes	Data
SortPicts.Native	StdCLib.105248	014FE6D0	00001F88	Yes	Data
SortPicts.Native	StdCLib.105248	0236C470	00010CF4	Yes	Code
SortPicts.Native	ThreadsLib.258704	01500680	0000020C	Yes	Data
SortPicts.Native	ThreadsLib.258704	02368F30	0000343C	Yes	Code
System	46DE40	0046E030	00000340	No	Code
System	46DE40	003639F0	00000080	No	Data
System	CodeFragmentMgr.409E08C0	0001FA60	000007CC	No	Data
System	CodeFragmentMgr.409E08C0	409E11B0	0000C2BC	No	Code
System	CodeFragmentMgr.D64E0#1	000A2930	00000894	No	Data
System	CodeFragmentMgr.D64E0#1	000D6F50	0000CA98	No	Code

Enter an address to display information.

00000000

Fragment:

Function:

In the Fragment Info window, each row represents a fragment. Each of the six columns displays information about the fragment, as described here.

- **Process Name.** The name of the process associated with this fragment. For each process, the window shows fragments containing the code for the process itself, as well as shared libraries that it calls. For example, in Figure 3-26, the `SortPicts.Native` process uses fragments belonging to the QuickDrawLib, StdCLib and ThreadsLib libraries, as well as its own code.
- **Fragment Name.** The name of the fragment. Several fragments can have the same name, but their addresses will be different. In Figure 3-26, for instance,

there are two fragments called `CodeFragmentMgr.409E08C0` and two called `CodeFragmentMgr.D64E0#1`. The addresses appended to the names show where the fragments reside in memory.

- **Address.** The address of the code or data. For a code fragment, the address is the same for all processes that use the fragment. For a data fragment, in contrast, each process that uses the fragment can get a separate copy of the fragment's global data; therefore, for data fragments there is a different starting address for each process.
- **Size.** The size of the fragment.
- **Dbg?** The target state of the fragment. Yes if it is currently targeted for debugging; no if it is not.
- **Type.** The type of the fragment (code or data).

By default, the items in the Fragment Info window are sorted alphabetically by process name. If you wish, you can sort the data by fragment name, address or size by clicking the heading at the top of the appropriate column.

When fragments are created or deleted on the target machine, the information in the Fragment Info window is not automatically updated. You can use the Refresh button to get an updated list at any time.

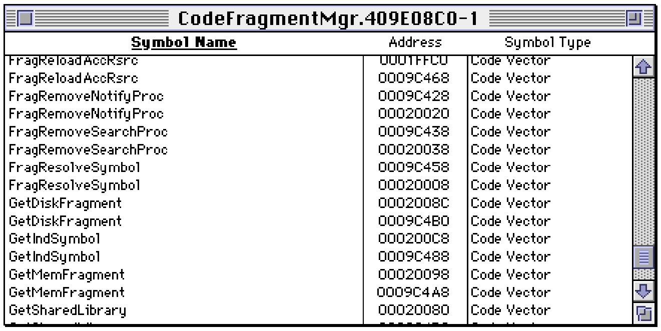
Sometimes you may want to know what code is located at a particular address in memory. If you enter that address in the text box at the bottom of the Fragment Info window, the display tells you which fragment, if any, is loaded at that address, and what function it is in, if the symbols are available.

You can also enter a transition vector into the text box to display the function name. See "Exports Windows" (page 79).

Exports Windows

A code fragment may contain routines that are exported for use by other fragments. To list these routines, double-click any row in the Fragment Info window or select the row and click the Show Exports button. An exports window appears, listing all the exports in the given fragment (Figure 3-27).

Figure 3-27 An exports window



CodeFragmentMgr.409E08C0-1

Symbol Name	Address	Symbol Type
FragReloadAccRsrc	0001FFC0	Code Vector
FragReloadAccRsrc	0009C468	Code Vector
FragRemoveNotifyProc	0009C428	Code Vector
FragRemoveNotifyProc	00020020	Code Vector
FragRemoveSearchProc	0009C438	Code Vector
FragRemoveSearchProc	00020038	Code Vector
FragResolveSymbol	0009C458	Code Vector
FragResolveSymbol	00020008	Code Vector
GetDiskFragment	0002008C	Code Vector
GetDiskFragment	0009C4B0	Code Vector
GetIndSymbol	000200C8	Code Vector
GetIndSymbol	0009C488	Code Vector
GetMemFragment	00020098	Code Vector
GetMemFragment	0009C4A8	Code Vector
GetSharedLibrary	00020080	Code Vector

In the window, each exported symbol is identified by name, address, and type (code vector or data). The address points to code or data, as follows:

- If the symbol type is code vector, the address is a pointer to the routine (a **transition vector**). Double-clicking the row brings up an instructions window showing the code for the function.
- If the symbol type is data, the address is a pointer to the data itself. Double-clicking the row brings up a memory window starting at that address.

By default, the exports list is sorted by symbol name. You can sort by address or symbol type by clicking the appropriate column header.

The Global Variables Window

The **Global Variables window** displays a list of global variables for all currently open symbol files. To open this window, choose Show Global Variables from the Window menu (or press Command-L). For more information, see “Global Variables” (page 101).

The Breakpoint List

The **Breakpoint List window** displays all the breakpoints that are currently set. To open this window, choose Show Breakpoint List from the View menu (or press Command-N). For more information, see “Setting Breakpoints” (page 85) and “The Breakpoint List” (page 94).

C H A P T E R 4

Basic Debugging Tasks

Contents

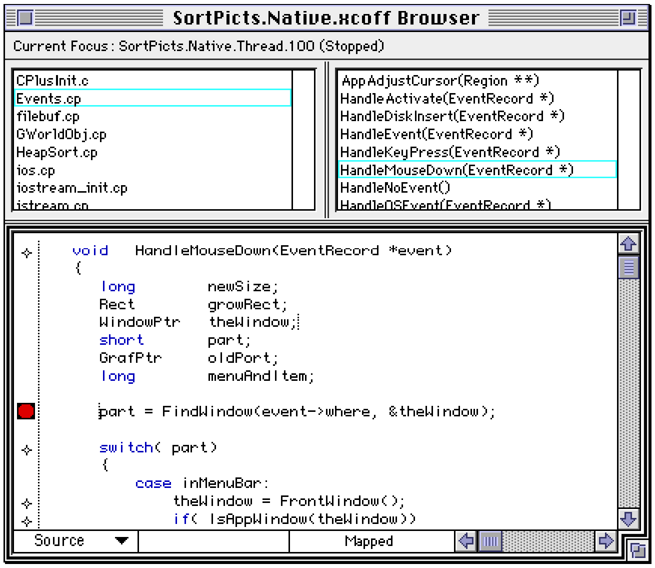
This chapter describes the fundamental tasks you perform when debugging: setting breakpoints, stepping through your code, and examining your variables.

Setting Breakpoints

A breakpoint stops execution of the target program at a specific statement or instruction. Once the program is stopped, you can examine the state of the system (such as memory, registers, and your program's variables).

You typically set breakpoints in the Code Browser window. You can also set a breakpoint in any other source-code or instructions window. Each of these windows contains a breakpoint column, the narrow space to the left of the vertical dotted line in a source-code or instructions window (see Figure 4-1).

Figure 4-1 Setting a simple breakpoint


SortPicts.Native.xcoff Browser
Current Focus : SortPicts.Native.Thread.100 (Stopped)
Source
Mapped

CPlusInit.c
Events.cp
filebuf.cp
GWorldObj.cp
HeapSort.cp
ios.cp
iostream_init.cp
istream.cp

AppAdjustCursor(Region **)
HandleActivate(EventRecord *)
HandleDiskInsert(EventRecord *)
HandleEvent(EventRecord *)
HandleKeyPress(EventRecord *)
HandleMouseDown(EventRecord *)
HandleNoEvent()
HandleOSEvent(EventRecord *)

```
void    HandleMouseDown(EventRecord *event)
{
    long        newSize;
    Rect        growRect;
    WindowPtr   theWindow;
    short       part;
    GrafPtr     oldPort;
    long        menuAndItem;

    part = FindWindow(event->where, &theWindow);

    switch( part)
    {
        case inMenuBar:
            theWindow = FrontWindow();
            if( IsAppWindow(theWindow))
```

- In the Browser or other source-code window, the hollow diamond icons in the breakpoint column show where you can set breakpoints. They correspond to the statements in your program.
- In any window displaying assembly code, you can set a breakpoint on any instruction. Therefore, no diamonds appear in the breakpoint column. Figure 3-12 (page 58) shows an example of a breakpoint in assembly code.

For more information about the windows in which you can set breakpoints, see “The Browser Window” (page 49) and “Instructions Windows” (page 57).

To set a breakpoint, click one of the diamonds; a breakpoint icon replaces the diamond. To set different types of breakpoints, you press one or more modifier keys while clicking. The cursor changes to reflect the type of breakpoint that is set. For some types of breakpoints, you need to enter additional information in a dialog box.

Note
If you attempt to set a breakpoint in ROM, the debugger displays an alert. ◆

Types of Breakpoints

The Power Mac Debugger allows you to set several types of breakpoints.

- A **simple breakpoint** stops every time it is encountered.
- A **one-shot breakpoint** stops once and is then removed. When you set a one-shot breakpoint, the program resumes execution immediately.
- A **counting breakpoint** stops after it is encountered a specified number of times.
- A **conditional breakpoint** stops when a specified condition evaluates to true.
- A **performance breakpoint** causes performance measurement to be turned on or off. You can choose whether to have the program stop when a performance breakpoint is reached.

In addition, any breakpoint can be a **focused breakpoint.** A focused breakpoint applies only to the process (or thread) that has the current focus at the time you set the breakpoint. Execution stops only if that process was executing when the breakpoint was hit. For example, if you set a focused breakpoint in a shared library, execution will stop only if the library is called by your application, not if it is called by any other program.

When you set a breakpoint, pressing the Command key (along with any other modifier keys) causes it to be a focused breakpoint. Table 4-1 shows the icons corresponding to each breakpoint and the modifier keys necessary to set them.

Table 4-1 Debugger breakpoint icons

Breakpoint	Icon	Modifiers
Simple	🟥	None
Focused Simple	🔴	Command
One-Shot	▼	Option
Focused One-Shot	▼	Command-Option
Counting	#	Control
Focused Counting	#	Command-Control
Conditional	?	Control
Focused Conditional	?	Command-Control
Performance On	P	Control
Performance On: Focused	P	Command-Control
Performance Off	P	Control
Performance Off: Focused	P	Command-Control

To delete a breakpoint, click its icon; the icon disappears and the breakpoint is removed.

The following sections give more detail about each type of breakpoint.

Simple Breakpoints

A simple breakpoint stops program execution every time it is encountered. To set a simple breakpoint, click once in the breakpoint column next to the statement you want to break on. A red octagon (stop-sign shape) inside a black square appears next to the statement, as shown in Figure 4-1 (page 85). If you set a focused simple breakpoint, its icon is a stop sign without a black square around it.

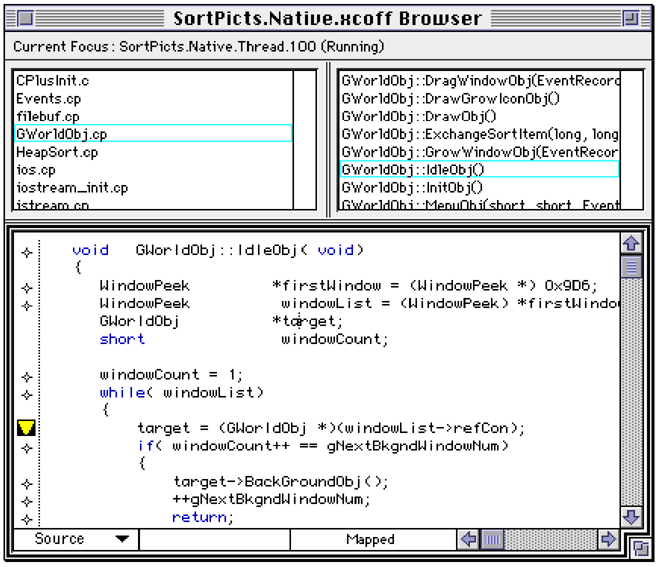
One-Shot Breakpoints

A one-shot breakpoint, or temporary breakpoint, allows you to set a breakpoint and resume execution immediately in one step.

To set a one-shot breakpoint, press Option while clicking in the breakpoint column. The debugger places a one-shot breakpoint icon where you clicked, resumes execution of the target application, and stops at the one-shot breakpoint (unless it hits another breakpoint first). When the one-shot breakpoint is reached, its icon is removed. The icon is a yellow yield sign inside a black square; for a focused one-shot breakpoint, it is a yellow yield sign only.

Figure 4-2 shows the Browser window after a one-shot breakpoint has been set.

Figure 4-2 Setting a one-shot breakpoint



SortPicts.Native.xcoff Browser

Current Focus : SortPicts.Native.Thread.100 (Running)

```
CPlusInit.c
Events.cp
filebuf.cp
GWorldObj.cp
HeapSort.cp
ios.cp
iostream_init.cp
istream.cp
```

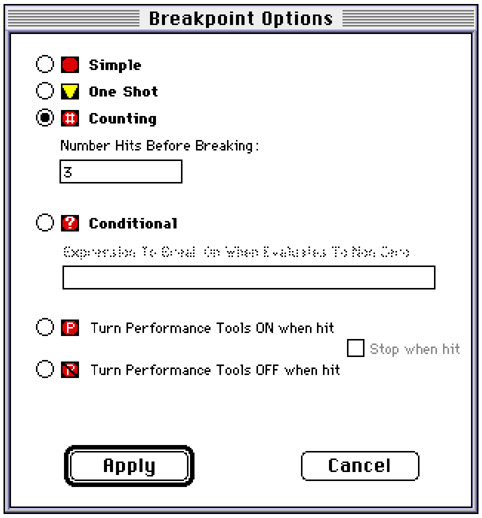
```
GWorldObj::DragWindowObj(EventRecord
GWorldObj::DrawGrowIconObj()
GWorldObj::DrawObj()
GWorldObj::ExchangeSortItem(long, long
GWorldObj::GrowWindowObj(EventRecor
GWorldObj::IdleObj()
GWorldObj::InitObj()
GWorldObj::MenuObj(short, short, Event
```

```
void   GWorldObj::IdleObj( void)
{
    WindowPeek          *firstWindow = (WindowPeek *) 0x9D6;
    WindowPeek           windowList = (WindowPeek) *firstWindo
    GWorldObj           *target;
    short                windowCount;

    windowCount = 1;
    while( windowList)
    {
        target = (GWorldObj *)(windowList->refCon);
        if( windowCount++ == gNextBkgndWindowNum)
        {
            target->BackGroundObj();
            ++gNextBkgndWindowNum;
            return;
```

The Breakpoint Options Dialog Box

If you press the Control key while clicking in the breakpoint column, the Breakpoint Options dialog box appears. (If you press Command and Control together, the dialog box is entitled Focused Breakpoint Options.) Using the Breakpoint Options dialog box, you can specify information needed to set counting, conditional and performance breakpoints. Figure 4-3 shows this dialog box.

Figure 4-3 Specifying breakpoint options



Breakpoint Options

○ Simple

○ One Shot

● Counting

Number Hits Before Breaking: 3

○ Conditional

Expression To Break On When Evaluates To Non Zero:

○ Turn Performance Tools ON when hit

☐ Stop when hit

○ Turn Performance Tools OFF when hit

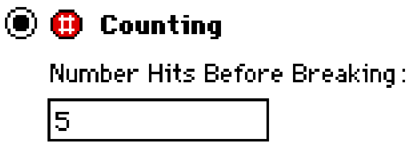
Note
You can use this dialog box to set any kind of breakpoint; however, it is easier to set a simple or one-shot breakpoint just by clicking in the breakpoint column. If you use the dialog box to set a one-shot breakpoint, the target does *not* resume immediately but waits for you to issue a Run command. ◆

Counting Breakpoints

A counting breakpoint halts program execution every *n*th time the breakpoint is encountered. You specify the value of *n* when you set the breakpoint.

To set a counting breakpoint, click the “Counting” radio button, as shown in Figure 4-4, and enter a number in the text box below it, then click Apply. A counting breakpoint icon appears next to the line of code for which the breakpoint has been set.

Figure 4-4 Selecting a counting breakpoint



Counting

Number Hits Before Breaking:

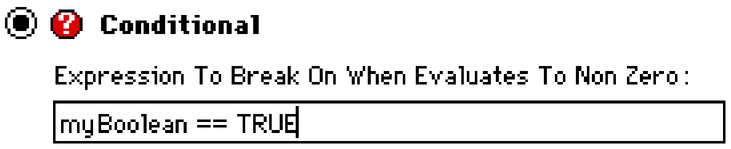
5

Conditional Breakpoints

A conditional breakpoint halts program execution when the breakpoint is encountered and a previously specified condition is true.

To set a conditional breakpoint, click the “Conditional” radio button, as shown in Figure 4-5, and enter an expression in the text box below, then click Apply. A conditional breakpoint icon appears next to the line of code where the breakpoint has been set.

Figure 4-5 Selecting a conditional breakpoint



Conditional

Expression To Break On When Evaluates To Non Zero:

myBoolean == TRUE

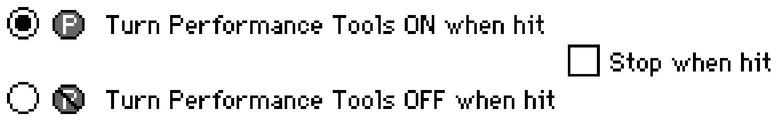
Note
For additional information about expression grammar, see Appendix B, “Expression Evaluation.” ◆

Performance Breakpoints

A performance breakpoint allows you to start or stop performance measurement for selected blocks of code.

To set a performance breakpoint, click one of the performance radio buttons illustrated in Figure 4-6. If you check the “Stop when hit” checkbox, the debugger stops when the breakpoint is reached; otherwise, performance measurement is turned on or off but the program keeps executing.

Figure 4-6 Setting a performance breakpoint



[✓ Turn Performance Tools ON when hit]

Turn Performance Tools OFF when hit

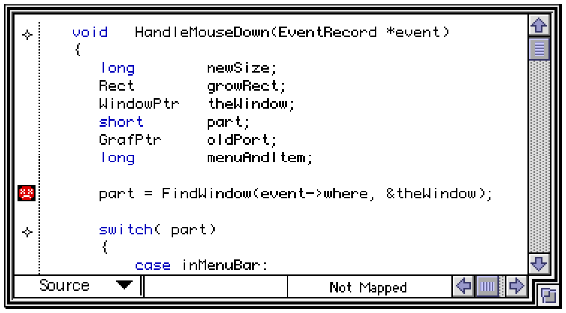
☐ Stop when hit

For more information about using the performance tools, see Chapter 6, “Measuring Performance.”

Inactive Breakpoints

If you set a breakpoint in code that is not mapped, the breakpoint icon has an “unhappy face” in it, as shown in Figure 4-7.

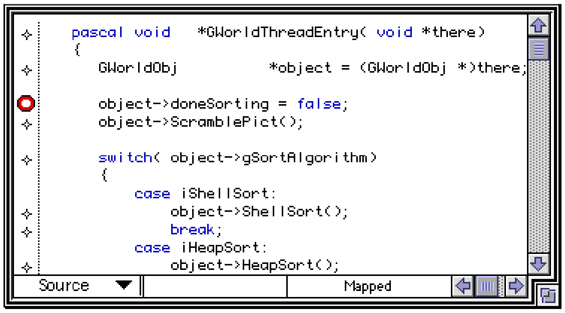
Figure 4-7 A breakpoint in unmapped code



```
void   HandleMouseDown(EventRecord *event)
{
    long        newSize;
    Rect        growRect;
    WindowPtr   theWindow;
    short       part;
    GrafPtr     oldPort;
    long        menuAndItem;

    part = FindWindow(event->where, &theWindow);

    switch( part)
    {
        case inMenuBar:
```

If you set a focused breakpoint, whenever the current focus is not the one that applies to this breakpoint, the icon appears as a hollow octagon, as shown in Figure 4-8.

Figure 4-8 A currently inactive focused breakpoint

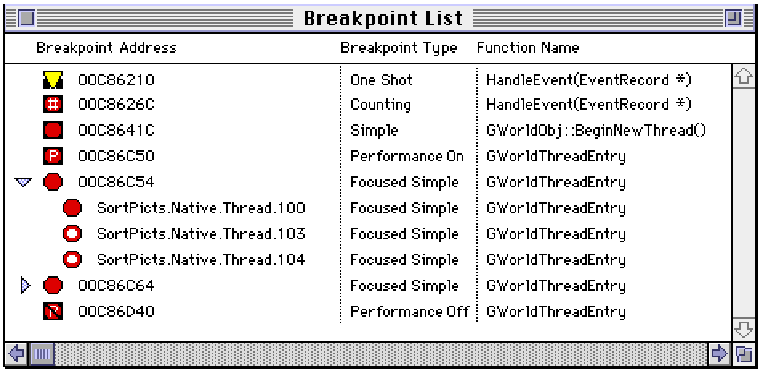


```
pascal void    *GWorldThreadEntry( void *there)
{
    GWorldObj           *object = (GWorldObj *)there;

    object->doneSorting = false;
    object->ScramblePict();

    switch( object->gSortAlgorithm)
    {
        case iShellSort:
            object->ShellSort();
            break;
        case iHeapSort:
            object->HeapSort();
```

The Breakpoint List

To see a list of the currently set breakpoints, choose Show Breakpoint List from the Window menu (or press Command-N). The debugger displays a window like the one shown in Figure 4-9.

Figure 4-9 Viewing the currently set breakpoints



Breakpoint List

Breakpoint Address	Breakpoint Type	Function Name
00C86210	One Shot	HandleEvent(EventRecord *)
00C8626C	Counting	HandleEvent(EventRecord *)
00C8641C	Simple	GWorldObj::BeginNewThread()
00C86C50	Performance On	GWorldThreadEntry
00C86C54	Focused Simple	GWorldThreadEntry
SortPicts.Native.Thread.100	Focused Simple	GWorldThreadEntry
SortPicts.Native.Thread.103	Focused Simple	GWorldThreadEntry
SortPicts.Native.Thread.104	Focused Simple	GWorldThreadEntry
00C86C64	Focused Simple	GWorldThreadEntry
00C86D40	Performance Off	GWorldThreadEntry

Each row represents a currently set breakpoint and shows the following information:

■ **The breakpoint's address.** If you double-click the address, an instructions window appears showing the assembly code for the function. A missing address means that the breakpoint has been set for code that is not yet targeted.

- **The breakpoint type.** If you double-click the type, the Breakpoint Options dialog box appears, allowing you to change the type of breakpoint. The icon representing the type is also displayed to the left of the address. Table 4-1 shows all the breakpoint icons and what they mean.

  For focused breakpoints, a disclosure triangle appears to the left of the icon; you can click it to reveal information about the context in which the breakpoint was set. You can have more than one focused breakpoint set at a given location, as shown in this example.

- **The function name,** if it is known. If you double-click the function name, the source view of the function is displayed in the Browser window.

  If the function name displayed is "????", it indicates that the debugger could not find symbolic information or embedded names mapped to the breakpoint's address. You cannot display source code for this function.

Removing Breakpoints

You can remove breakpoints one at a time or you can remove several at once. There are two ways to remove a single breakpoint:

- Click the breakpoint's icon in a code view.
- Select the breakpoint in the Breakpoint list and press Delete.

Here's how to remove several breakpoints at once:

- Select the breakpoints in the Breakpoint list and press Delete. (To select these breakpoints, Shift-click on their rows.)
- To remove all breakpoints, choose Clear All Breakpoints from the Control menu.

Setting Breakpoints on System Calls

The debugger has no mechanism for setting breakpoints on A-traps. On PowerPC-based Mac OS systems, however, all system software routines are called through glue routines. In the Browser, the list of source files includes an entry named "•••synthesized glue•••". This is not actually one of your source files; rather, it lists all the glue routines used by your code. There is a glue routine for each imported routine, including Toolbox and operating system calls, as well as shared library calls. As a result, you can set breakpoints in the appropriate glue routines to catch calls into system or library routines.

Entering the Debugger From Your Source Code

In addition to setting breakpoints while debugging, you can put calls to two system routines, `Debugger` and `DebugStr`, into your source code in order to determine when to enter the debugger. To use these routines, include the header file `Types.h` in your code.

- When the program hits a `Debugger` routine, the debugger is entered. The program counter points to the next instruction to be executed.

  The syntax of the `Debugger` routine is

  ```
  Debugger();
  ```

- When the program hits a `DebugStr` routine, the debugger is entered. The debugger writes a string (the parameter to the `DebugStr` routine) to the log window, and the program counter points to the next instruction to be executed.

  Here is an example of a call to `DebugStr`:

  ```
  DebugStr((ConstStr255Param)"\pOutput this string to the Log
                window");
  ```

If, instead, you want the debugger to ignore these calls, then uncheck the “Set stop on Debugger()/DebugStr() default” option from General Preferences in the Edit menu. Also, you can control this option on a process basis by using the Process Browser; see “Application Preferences” (page 77).

Controlling Program Execution

This section discusses how to control the execution of your program. In “Program Control Commands” (page 61), you saw the commands that are available from the Control Palette, along with their icons. This section describes these commands further. In addition, it describes other commands that are accessible only from the Control menu (Figure 4-12).

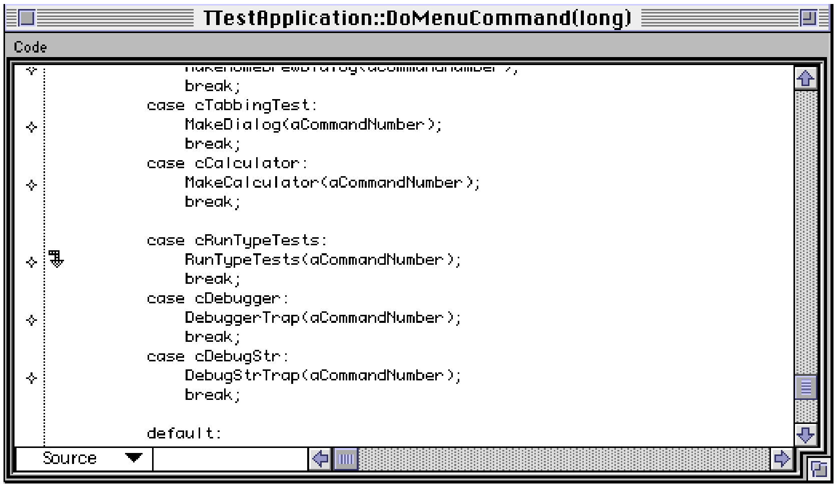
The Current Statement

When your code is stopped, the program counter (PC) is displayed as a solid arrow to the left of the current statement or instruction, as shown in Figure 3-8.

If the arrow is not visible (if, for instance, it is in a window that has been obscured), you can make it visible by choosing Current PC from the View menu. The current routine is normally displayed in the Browser window. If the PC is executing code that is outside your source code, it is shown in an instructions window.

If you use the Browser (or other code window) to display a routine other than the current routine, but which is part of its call chain, a gray downward-pointing arrow is shown at the last statement that was executed. This arrow can be seen in Figure 4-10, in which the routine `TTestApplication::DoMenuCommand` is displayed. In it, the arrow indicates that the routine `RunTypeTests` was called and that the program stopped somewhere in that routine (or a routine that it in turn called).

Figure 4-10 The PC in subroutine marker



```
            break;
        case cTabbingTest:
            MakeDialog(aCommandNumber);
            break;
        case cCalculator:
            MakeCalculator(aCommandNumber);
            break;

        case cRunTypeTests:
            RunTypeTests(aCommandNumber);
            break;
        case cDebugger:
            DebuggerTrap(aCommandNumber);
            break;
        case cDebugStr:
            DebugStrTrap(aCommandNumber);
            break;

        default:
```

To see the entire call chain, you can show the Stack Crawl window. For more information, see “Navigating the Call Chain” (page 64).

Stepping Through Your Code

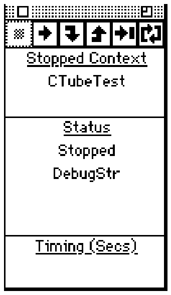
The Power Mac Debugger provides several methods of stepping through your code. To step through code, the following conditions must be met:

- The code must be targeted. This means the debugger is aware of it.
- The code must have the current focus. If more than one process is targeted, only one can be the current focus.
- The code must be stopped.

For more information on targeting, focusing and stopping processes, see "Targeting and Stopping Your Code" (page 40) and "The Process Browser" (page 73).

When you use the serial nub, a stopped application is referred to as the stopped context, as shown in Figure 4-11; it may or may not be the same as the current focus. The stopped context is the only process in which you can single-step. The current focus, if different, is used only for global evaluation; see "Global Variables" (page 101).

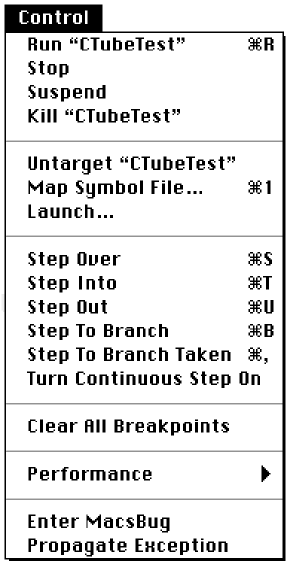
Figure 4-11 The stopped context in the serial nub



If the frontmost code window is showing a source view, stepping proceeds one source-code statement at a time. If the window is showing an assembly view, stepping proceeds one assembly instruction at a time.

The commands you use to step through your code are available from the Control menu, as shown in Figure 4-12. Many of them are available from the control palette as well.

Figure 4-12 The Control menu



Choose Step Over from the Control menu (or press Command-S) to execute the next statement. If the next statement is a subroutine call, Step Over causes the subroutine to be executed in its entirety and control stops at the statement following the subroutine call. In other words, you "step over" the subroutine.

If the next statement is a subroutine call, and you want to single-step through the subroutine, choose Step Into from the Control menu (or press Command-T). This command causes execution to proceed with the first statement of the subroutine. If you step into a routine that is not part of your program (for instance, a system call) the debugger brings up an instruction display window showing the assembly code for that routine; see "Instructions Windows" (page 57). You can then continue to step through the code, at the assembly-instruction level, if you wish.

If you want to exit the routine and return to your source code, you can use the Step Out command (or press Command-U). This causes execution to proceed to the statement just following the one that called the subroutine. This process is useful any time you want to exit from a subroutine.

Note
When you initiate a Step Out command, the debugger implements it by placing a one-shot breakpoint at the statement after the calling statement. Typically, this breakpoint appears and disappears quickly; however, if you hit a breakpoint while stepping out, the one-shot breakpoint is still in effect (it also appears in the Breakpoint List). See “One-Shot Breakpoints” (page 89) for more information. ◆

If you choose Step To Branch from the Control menu (or press Command-B), the debugger steps over all instructions until it reaches any branch instruction, at which point it stops. Step To Branch Taken (Command-comma) has the same effect as Step To Branch, except that it stops only if the branch is actually going to be taken. While these two commands are most meaningful when viewing assembly code, they can also be used when viewing source code.

If you want to execute one of the step commands repeatedly, so that you can watch your program execute in “slow motion”, choose Turn Continuous Step On from the Control menu or click the corresponding icon in the control palette. Then choose Step, Step Into, or Step Out, and the program steps repeatedly until it hits a breakpoint. To stop the continuous stepping, choose Turn Continuous Step Off or click the control palette icon again.

Other Program Control Commands

In addition to the “step” commands, the debugger has other commands for controlling your program. All of these are reached through the Control menu.

Choose Run (or press Command-R) to continue execution of the program with the current focus. If this program is already running, it is brought to the foreground.

Choose Stop to stop the program with the current focus. This command is available only when using the serial nub.

Choose Kill to terminate the process currently selected in the Process Browser. All of the windows associated with the process are closed.

Choose Suspend to suspend execution of a thread currently selected in the Process Browser. You can resume execution by selecting the thread and choosing Resume from the Control menu.

Choose Launch to launch an application and have it stop before executing `main`. The Launch command only works for one-machine debugging; to break on launch when using two machines, hold down the Control key on the target machine as you double-click the application.

The Power Mac Debugger is entered whenever an application generates an exception (such as a `DebugStr` call). If you choose Propagate Exception, control passes to the next exception handler that exists on the system. Typically, this is a low-level debugger such as MacsBug.

In addition, you can choose Enter MacsBug from the Control menu to go directly to MacsBug (or another low-level debugger that is installed on your target machine).

Note
Enter MacsBug is slightly different from Propagate Exception in that it generates a separate exception as opposed to propagating the existing one. ◆

Besides the Control menu commands, there is one other way to alter your program's execution: you can click the PC arrow and drag it to another statement. When you resume the program, execution begins where the new PC is. Be extremely careful when using this option, because you can get unexpected (and possibly fatal) results when executing statements out of order.

Displaying and Editing Variables

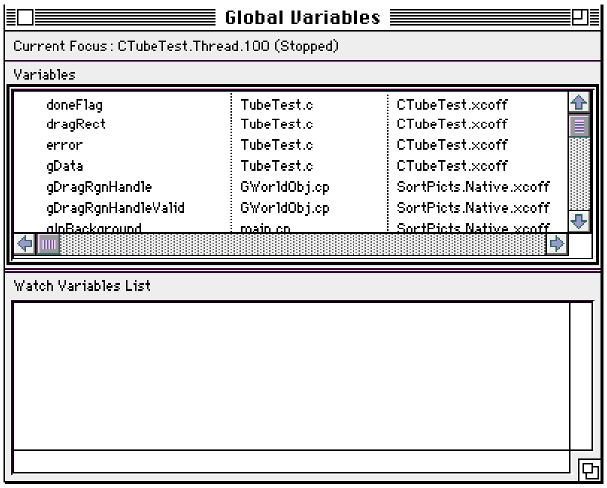
This section describes some of the ways you can examine your program's variables. See also "Examining Local Variables" (page 65) for information on using the Stack Crawl to see local variables.

Global Variables

To display the names and values of your program's global variables, choose Show Global Variables from the Window menu (or press Command-L). The debugger opens a two-pane window like the one shown in Figure 4-13.

The top pane of the Global Variables window lists the global and static variables for all programs for which a symbol file is open, in alphabetical order. Each row has three columns, showing the variable's name, the source file in which it is declared and the corresponding symbol file. The columns can be resized just as the window panes can.

Figure 4-13 The Global Variables window



Global Variables

Current Focus : CTubeTest.Thread.100 (Stopped)

Variables

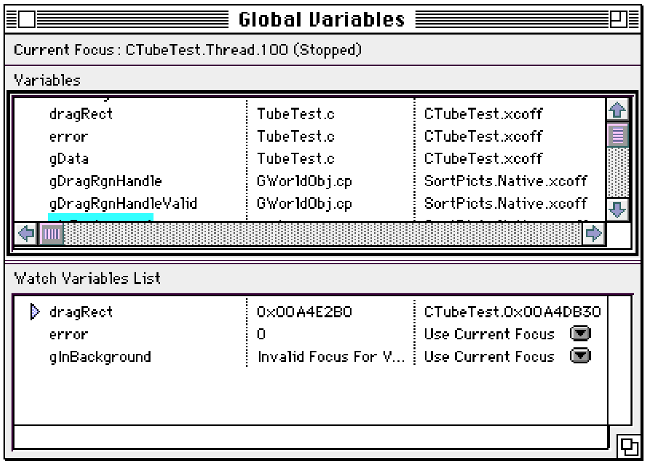
doneFlag	TubeTest.c	CTubeTest.xcoff
dragRect	TubeTest.c	CTubeTest.xcoff
error	TubeTest.c	CTubeTest.xcoff
gData	TubeTest.c	CTubeTest.xcoff
gDragRgnHandle	GWorldObj.cp	SortPicts.Native.xcoff
gDragRgnHandleValid	GWorldObj.cp	SortPicts.Native.xcoff
glpBackground	main.cp	SortPicts.Native.xcoff

Watch Variables List

Note
Some global variables may live in “data-only” fragments. To display the values of these variables, you must open a symbol file corresponding to their fragment. Because the fragment contains data only, the Browser window that opens does not display any functions. ◆

To see the value of a variable, double-click anywhere in its row. An entry for the variable appears in the lower pane, the Watch Variables List, as shown in Figure 4-14. To remove an item from the Watch Variables List, select it and press Delete.

Figure 4-14 Viewing a global variable's value



dragRect	TubeTest.c	CTubeTest.xcoff
error	TubeTest.c	CTubeTest.xcoff
gData	TubeTest.c	CTubeTest.xcoff
gDragRgnHandle	GWorldObj.cp	SortPicts.Native.xcoff
gDragRgnHandleValid	GWorldObj.cp	SortPicts.Native.xcoff

dragRect	0x00A4E2B0	CTubeTest.0x00A4DB30
error	0	Use Current Focus
gInBackground	Invalid Focus For V...	Use Current Focus

The Watch Variables List also has three columns, showing the variable's name, its value, and the context in which it is being evaluated. This context is, by default, the thread or process that has the current focus. A pop-up menu allows you to change it to any context to which the variable applies. In the case of a shared library, for example, a global variable may be instantiated several times, once for each application that uses the library.

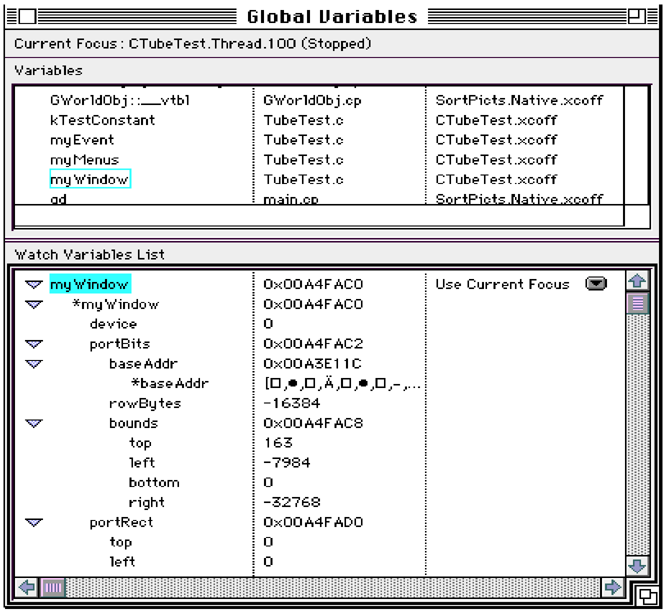
If the global variable does not exist in the current focus, the value is shown as "Invalid Focus for Variable!" For example, in Figure 4-14 the variable `error` belongs to the application CTubeTest, which is not the current focus. To show the value, use the pop-up menu and select an appropriate context. In Figure 4-14, CTubeTest was selected as the context for the global variable `dragRect`.

Note
A global variable may exist in no context (when the code corresponding to a symbol file is not running). In this case, it is not possible to display the value of the variable. ◆

Global variables are displayed in the same way that local variables are displayed in the Stack Crawl window. Note that some variable names have disclosure triangles to their left and some do not.

- Simple (nonstructured) data types or arrays are shown without disclosure triangles. Their value is displayed to the right of their name.
- Structured data types, pointers, and handles use disclosure triangles to allow you to get more information.
  - If the variable is a structured data type, the value displayed represents its address. (All addresses are displayed in hexadecimal, with the prefix "0x".) You can see the structure's fields by clicking the disclosure triangle to the left of its name. Any fields that are structures or pointers are displayed in turn using disclosure triangles, as shown in Figure 4-15.
  - If the variable is a pointer (or a handle), you can click its disclosure triangle to display its dereferenced value. The next line will show the name of the pointer preceded by an asterisk (*). The value displayed follows the same rules as for other variables: simple variables are displayed directly; structures are displayed showing their address, and can be expanded further to show their fields.

Figure 4-15 Examining global variables


Global Variables

Current Focus : CTubeTest.Thread.100 (Stopped)

Variables

```
GWorldObj::__vtbl      GWorldObj.cp     SortPicts.Native.xcoff
kTestConstant          TubeTest.c       CTubeTest.xcoff
myEvent                TubeTest.c       CTubeTest.xcoff
myMenus                TubeTest.c       CTubeTest.xcoff
myWindow               TubeTest.c       CTubeTest.xcoff
qd                     main.cp          SortPicts.Native.xcoff
```

Watch Variables List

```
▽ myWindow               0x00A4FAC0       Use Current Focus
▽    *myWindow           0x00A4FAC0
       device            0
▽      portBits          0x00A4FAC2
▽        baseAddr        0x00A3E11C
            *baseAddr    [□,●,□,Ä,□,●,□,-,...
         rowBytes        -16384
▽        bounds          0x00A4FAC8
           top           163
           left          -7984
           bottom        0
           right         -32768
▽      portRect          0x00A4FAD0
         top             0
         left            0
```

If you double-click the name of a variable that has a disclosure triangle, an expression results window opens, displaying the contents of the variable in expanded form. If the window is left open, its information is updated whenever the program stops. See “Expression Results Windows” (page 105) for more information.

Just as with Browser windows, you can create a “clone” of the Watch Variables List by holding the Option key down, clicking in the pane, and dragging. A new window is created showing the global variables you selected. Their values are updated whenever a program stops.

You can change the value of a global variable by selecting its value and typing the new value. The value takes effect when you press Return, Enter, or Tab, or click in a new field. You can undo the change by choosing Undo from the Edit menu (or pressing Command-Z). As always, remember to use caution when changing the value of any variable.

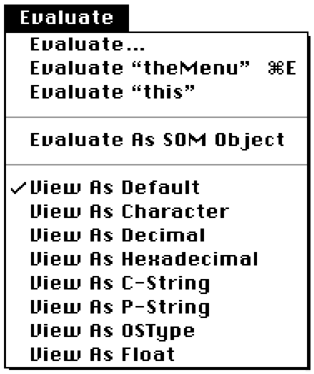
Expression Results Windows

The result of any evaluation (such as a variable or arithmetic expression) can be displayed in its own window. The title of the window is the expression (in this case, a variable) itself.

There are several ways to create expression results windows:

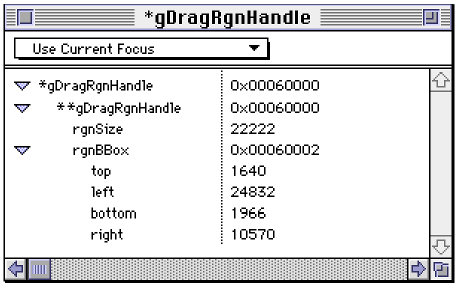
- You can double-click the name of a variable in the Stack Crawl or Global Variables windows. See “Examining Local Variables” (page 65) and “Global Variables” (page 101) for more information.
- You can choose the Evaluate command from the Evaluate menu (Figure 4-16), then enter the value of the variable or expression.

Figure 4-16 The Evaluate menu



- You can select a variable or expression name in a code view and choose Evaluate Selection from the Evaluate menu (or press Command-E). The word "Selection" is replaced by the selection, if there is one. For instance, if you select the variable `MyCounters`, the menu item reads Evaluate "MyCounters".

Figure 4-17 An expression results window



```
▽ *gDragRgnHandle          0x00060000
▽   **gDragRgnHandle       0x00060000
      rgnSize              22222
▽     rgnBBox              0x00060002
        top                1640
        left               24832
        bottom             1966
        right              10570
```

The display in Figure 4-17 contains several disclosure triangles, which allow pointers to be dereferenced and structures to be expanded. By default, variables are automatically expanded (that is, the disclosure triangles are turned downward) up to three levels. If there is further nesting, you can do the expansion manually by clicking the triangles. Choose General Preferences in the Edit menu to change the default number of auto-expansions. See Appendix A, “Debugger Preferences,” for more information.

The pop-up menu at the top of the window allows you to change the context in which a variable is being evaluated, in the same way as it is done in the Watch Variables pane of the Global Variables window.

To change the type of a value in an expression results window, select the value and choose one of the options shown in Table 4-2 from the Evaluate menu. If an option is not appropriate for the given expression or variable, it is dimmed.

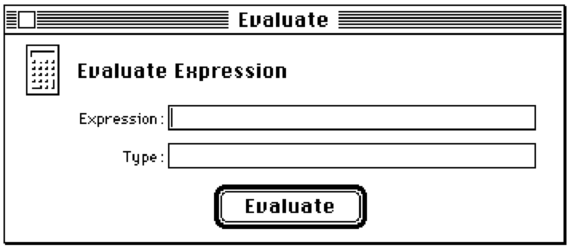
Table 4-2 Changing the type of a displayed expression

Item	**Effect**
View As Default	Display the selected value using the appropriate default format. The default format for numbers is decimal; for unsigned longs, it is decimal unless you choose “View unsigned long as hex by default” in the General Preferences dialog box. The default format for strings is an array of type `char`.
View As Character	Display the selected value as a character.
View As Decimal	Display the selected value as a signed decimal number.
View As Hexadecimal	Display the selected value as a hexadecimal number.
View As C-String	Display the selected value as a C string.
View As P-String	Display the selected value as a Pascal string.
View As OSType	Display the selected value as a four-character literal.
View As Float	Display the selected value in scientific notation.

Using the Evaluate Dialog Box

If you choose Evaluate from the Evaluate menu, the debugger displays a dialog box like the one shown in Figure 4-18.

Figure 4-18 The Evaluate dialog box



In the Type text field, you can enter any valid C or Mac OS data type, such as `short`, `char*`, or `OSErr`.

In the Expression text field, you can enter

- the name of a variable. When you click Evaluate, an expression results window appears, showing the variable and its value. If you specify a type, the debugger attempts to coerce the variable to that type.
- an address. In this case you must enter a type name in the Type text field so that the data can be interpreted correctly. The type entered must be some type of pointer; otherwise, it is evaluated as a plain number. For example, you might enter the type as `WindowRecord*`; the window displays the contents of a WindowRecord beginning at the address you specify.
- an expression.

  Here are some examples of valid expressions:

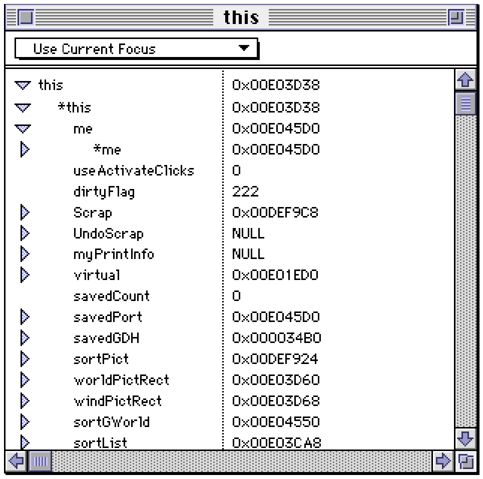
```
&a[0] != ptrs[0]
0xabc + 1234L - r1 + fp0              /* r1 specifies GPR 1 */
                                      /* fp0 specifies FPR 0 */
sizeof(\"abcdef\")
$abc + #12L + fp0                     /* $ specifies hexadecimal */
                                      /* # specifies decimal */
(b << 4) >> 8
(*(**Handle)->substruct)
 '\\n' + '\\t' + '\\v' + '\\0123' - '\\x1fa' - 'abc'
```

  For additional information about the evaluation of expressions, see Appendix B, “Expression Evaluation.”

Evaluating “this”

If the target program is stopped in a C++ method, you can display the value of the instance of the current class (that is, the object whose method is being executed) by choosing Evaluate “this” from the Evaluate menu. The fields of the object are shown in a window like the one in Figure 4-19. This value is also available in the Variables pane of the Stack Crawl window.

Figure 4-19 Evaluating “this”



```
this                    0x00E03D38
  *this                 0x00E03D38
    me                  0x00E045D0
      *me               0x00E045D0
    useActivateClicks   0
    dirtyFlag           222
    Scrap               0x00DEF9C8
    UndoScrap           NULL
    myPrintInfo         NULL
    virtual             0x00E01ED0
    savedCount          0
    savedPort           0x00E045D0
    savedGDH            0x000034B0
    sortPict            0x00DEF924
    worldPictRect       0x00E03D60
    windPictRect        0x00E03D68
    sortGWorld          0x00E04550
    sortList            0x00E03CA8
```

Note
The Evaluate “this” command is enabled only when the Browser is the frontmost window and the code pane is selected, or when another code view is the frontmost window. ◆

Evaluating SOM Objects

IBM’s **System Object Model™ (SOM™)** is a technology that allows programs to use object classes written in any programming language without the need for recompilation. The Power Mac Debugger can display, to a limited degree, the contents of SOM objects referenced in your programs.

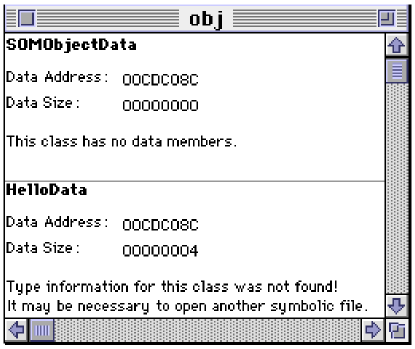
To display the contents of SOM objects, you must

- have installed the system extension SOMobjects™ for Mac OS (also referred to as the SOM kernel).
- open all symbol files that define type information for the SOM objects you wish to display.

  If the symbol files are not open or available, the debugger can display only address information, not the names of fields.
- select the name of a SOM object in a code view and choose Evaluate As SOM Object from the Evaluate menu.

In Figure 4-20, the object being evaluated doesn't define any data members, but its superclass does.

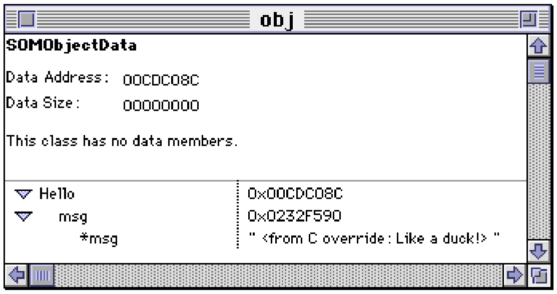
Figure 4-20 SOM object display



```
SOMObjectData

Data Address:  00CDC08C
Data Size:     00000000

This class has no data members.

HelloData

Data Address:  00CDC08C
Data Size:     00000004

Type information for this class was not found!
It may be necessary to open another symbolic file.
```

Figure 4-21 shows the result of the evaluation when the symbol file for the superclass has been opened.

Figure 4-21 SOM object with superclass data



obj

SOMObjectData

```
Data Address:  00CDC08C
Data Size:     00000000

This class has no data members.

▽ Hello          0x00CDC08C
▽   msg          0x0232F590
      *msg       " <from C override: Like a duck!> "
```

C H A P T E R 5

Advanced Debugging

Contents

This chapter contains specialized information that not all developers will need.

Debugging Non-Application Code

Whenever you debug code using the Power Mac Debugger, you are operating within the context of a process, or a thread within a process. As described in "The Current Focus" (page 76), the process or thread being debugged is referred to as the current focus. The Process Browser lists all the targetable processes on the target machine. In addition to targeting applications, you can also target the system process.

When you want to debug code that is not packaged as an application (shared libraries, drivers, extensions, MPW tools, and stand-alone code resources), you must take certain steps to ensure that the correct process is targeted.

To make sure your code is targeted correctly, you can use `DebugStr` or `Debugger` calls. When one of these calls is executed, the debugger takes control and automatically targets the correct process. If you are debugging a driver or extension, the system process may be the target. You will then be able to set breakpoints and debug your code.

If you do not use `DebugStr` or `Debugger` calls, you must make sure that the process in which your code is running is targeted. If the code is loaded in the system heap, you must target the system process explicitly.

The following sections give specific hints for debugging shared libraries and MPW tools.

Shared Libraries

A **shared library** is a code fragment of type `'shlb'` containing code that can be called by one or more other fragments. This code is loaded into memory only once regardless of how many fragments use it, although each caller (context) has its own copy of the library's global variables.

To debug a shared library, you must

- launch and target an application that calls the shared library. (It is not necessary to open a symbol file for this application, although it may be helpful.)

- open a symbol file for the library. Once you have done this, the status panel in the Browser window for the shared library should read “Mapped”.

Once the library has been targeted, you can begin performing operations such as setting breakpoints in the library, do single-stepping, and so on.

Note
If you set a focused breakpoint in the shared library, execution stops there only when the currently focused application calls the shared library. Any other calls to the shared library do not cause a break. ◆

MPW Tools

To debug an MPW tool written in native PowerPC code,

- open a symbol file for the tool
- hold down the Control key while executing the tool from the MPW shell command line

  This causes a break on launch in the tool. You can then set breakpoints in the tool’s Browser window and continue debugging.

Assembly-Level Debugging Without a Symbol File

If you do not have a symbol file, you can still debug your code at the assembly level. If the code has been compiled using a directive that generates embedded symbols (also known as MacsBug symbols), the Power Mac Debugger will display the symbol names in instructions windows and stack crawls, as well as in performance reports.

Debugging Emulated Code

The Power Mac Debugger has limited ability to debug emulated 680x0 code running on a PowerPC-based Mac OS computer. Here are the operations available to you:

- You can disassemble any area of memory as 680x0 instructions. See "Disassembling 680x0 Code" (page 59).
- You can see the addresses of 680x0 routines in the calling chain. See "Navigating the Call Chain" (page 64).

You cannot, however, use the Power Mac Debugger to step through 680x0 code or set breakpoints in it. For these operations, you can use MacsBug (or a similar low-level debugger). Once you know the address of an emulated routine you want to debug, you can choose Enter MacsBug from the Control menu. Then you can use commands such as `br` to set breakpoints and `so` to step through code. You can type `g` (GO) to return to the Power Mac Debugger.

Using a ROM Map

The debugger can use a ROM map to identify memory ranges in ROM with symbol names. This map allows you to see symbol names in ROM disassemblies and in ASP performance reports.

If you choose Open ROM Map from the File menu, a standard file dialog box allows you to select a file (usually called RomInfo) to use as the ROM map; the debugger then uses this file every time you launch it.

C H A P T E R 6

Measuring Performance

Contents

When you are trying to optimize the performance of a program, it is useful to know how much time is spent executing different sections of code. This can help you see where to focus your efforts for best results.

This chapter explains how you can use the Power Mac Debugger to analyze the performance of a program, using a utility called the **Adaptive Sampling Profiler (ASP)**. The first two sections, "About the Adaptive Sampling Profiler" and "Using the Adaptive Sampling Profiler," explain what the ASP is and how to use it. The last section, "How the ASP Gathers Data," is not essential but gives extra information that may be useful in interpreting the performance results.

About the Adaptive Sampling Profiler

To measure performance, the ASP samples the program counter (PC) at regular intervals; that is, it determines where the PC is. To keep track of the sampling information, the ASP divides the computer's memory into several discrete ranges, or **buckets**. When it takes a sample, the ASP determines which bucket the PC falls in and increments the count of hits for each bucket.

Instead of associating a fixed-size region of memory with each bucket, as most sampling tools do, the ASP dynamically adjusts the sizes of the buckets as it takes the samples (hence the term *adaptive*). The goal is to give you the finest granularity for those routines or instructions that execute most often. See "How the ASP Gathers Data" (page 134) for details on how buckets are allocated.

After you have run your program for a while, you generate a report that displays the sampling results. The report shows the amount of time spent in each routine in your application, as well as the time spent in other code running on the system (for instance, Toolbox routines). The ASP uses your symbol files to associate memory ranges with specific routines in your code.

In addition to applications, you can measure the performance of code in shared libraries or in stand-alone code resources. To do so, you must have an application that calls routines in the shared library or that loads and executes the routines in the stand-alone code resource.

Note
You cannot use the ASP to measure the performance of code that runs at interrupt time. ◆

Using the Adaptive Sampling Profiler

Because the ASP is an integral part of the Power Mac Debugger, no additional files are required to use it. In addition, unlike some other performance tools, the ASP does not require you to modify your source code to use it.

Before you can use the ASP, you must

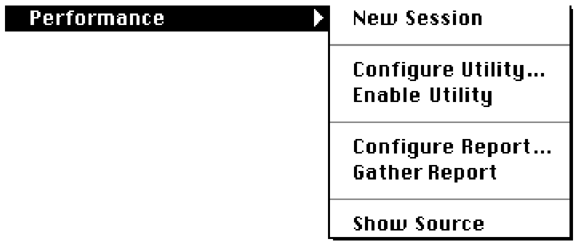
- launch the Power Mac Debugger
- launch the application to be measured
- open the symbol file or files for the code you are measuring

You can use the ASP on a one-machine or two-machine setup, with either the application nub or the serial nub.

Before you can start a sampling session, the application must initially be stopped. You can stop your application by using one of the methods described in Chapter 2, “Getting Started,” such as on launch, inserting a `DebugStr` call in your code, or setting a breakpoint.

All the commands you use when taking performance measurements are found in the Performance submenu in the Control menu (see Figure 6-1).

Figure 6-1 The Performance submenu



Normally, you take the following steps when using the ASP:

1. **Create a performance window (New Session).**

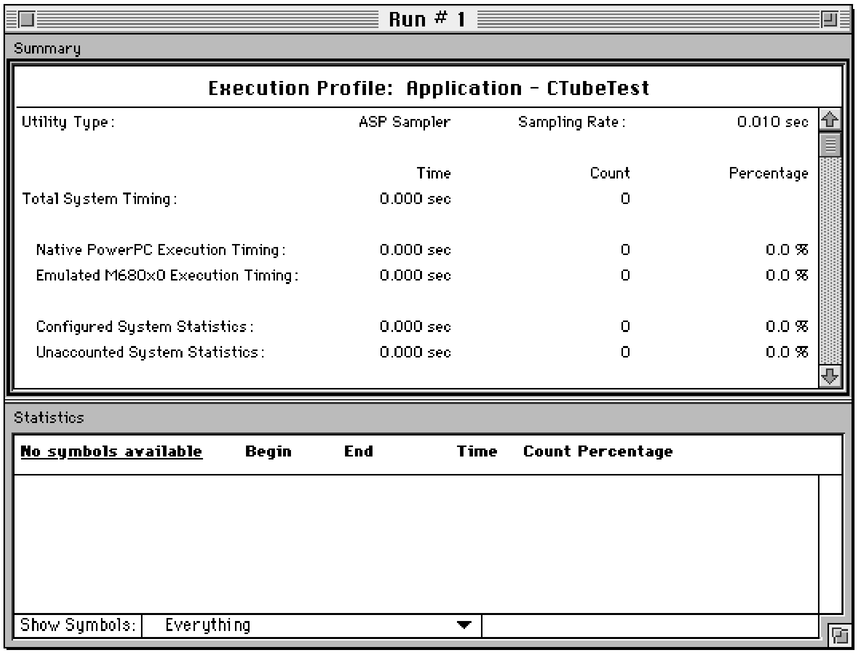
2. **(Optionally) set the sampling rate (Configure Utility).**
3. **Turn on the ASP (Enable Utility).**
4. **Run the program you are measuring.**
5. **Stop the program, or turn off the ASP (Disable Utility).**
6. **Display the statistics (Gather Report).**
7. **(Optionally) filter out data from the report (Configure Report).**
8. **(Optionally) display sampled routines (Show Source).**

The following sections describe the steps in more detail.

Starting a Profiling Session

To begin a profiling session, choose New Session from the Performance submenu. A **performance window** that will display the performance data for this session is displayed (Figure 6-2). Performance windows are numbered consecutively: the first one is named Run #1, the second is Run #2, and so forth. Since only one performance window can be open at a time, to begin a new session you must close any existing performance window.

Figure 6-2 A blank performance window



```
Run # 1
Summary
                 Execution Profile:  Application - CTubeTest
Utility Type:                           ASP Sampler     Sampling Rate:        0.010 sec

                                               Time              Count        Percentage
Total System Timing:                      0.000 sec                  0

  Native PowerPC Execution Timing:        0.000 sec                  0            0.0 %
  Emulated M680x0 Execution Timing:       0.000 sec                  0            0.0 %

  Configured System Statistics:           0.000 sec                  0            0.0 %
  Unaccounted System Statistics:          0.000 sec                  0            0.0 %

Statistics
No symbols available    Begin      End        Time    Count Percentage

Show Symbols:  Everything
```

The performance window contains two scrollable panes: the summary view and the statistics view. At the top of the summary view is the name of the fragment that contains the `main` function for the current process (in this example, CTubeTest). After you run your code and gather the report, the data is displayed in these panes. For more information about the information in these panes, see “Generating a Performance Report” (page 127).

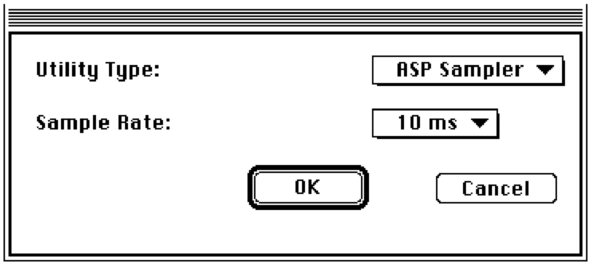
You can change the relative size of these two views by placing the cursor directly over the split lines and dragging the resize icon up or down to enlarge the desired view.

Specifying a Sampling Rate

At regular intervals, the ASP issues an interrupt and records the current PC value. This interval is called the **sampling rate**; the default value is 10 milliseconds (ms).The sampling rate for the current session is shown in the upper-right corner of the performance window.

To specify a different sampling rate, choose Configure Utility from the Performance submenu. A dialog box like the one shown in Figure 6-3 is displayed.

Figure 6-3 Setting the sampling rate


Utility Type: ASP Sampler
Sample Rate: 10 ms

The dialog box has two pop-up menus:

- Utility Type (currently the ASP Sampler is the only utility available)
- Sample Rate (you can choose here a value between 1 and 20 ms)

Begin by using the default sampling rate. You may then wish to try different settings to arrive at the best overall picture of your program's performance. Keep in mind that a time interval that is too large can result in a partial view of your program's performance. In contrast, an interval that is too small can result in excessive interrupt processing that might distort the results.

Once you have resumed execution with the ASP enabled, you cannot change the sampling rate.

Collecting Performance Data

To begin measuring the performance of your code, follow these steps:

1. **Choose Enable Utility from the Performance submenu.**

This command changes to read Enable ASP Sampler after you create a new session. After you choose it, the command then changes to read Disable ASP Sampler.

2. **Resume execution of your application.**

   You can resume your application by

   - choosing Run from the Control menu or the control palette. The ASP then begins taking sampling measurements.
   - using the Step commands (Step Over, Step Into, Step Out). The ASP measures performance during the time the code is actually running, not during the time you are stopped in the debugger.

3. **Exercise the features of your application.**

   Test those areas of your application that you are interested in, to be sure the results of the performance measurement are meaningful.

4. **Stop your program.**

   While the ASP is collecting data, no new information appears in the performance window. This window is updated only when you stop measurement and generate a performance report.

5. **Choose Gather Report from the Performance submenu.**

   The ASP generates a report for the data collected so far. See "Generating a Performance Report" (page 127) for information on interpreting the report.

Measuring Selected Routines

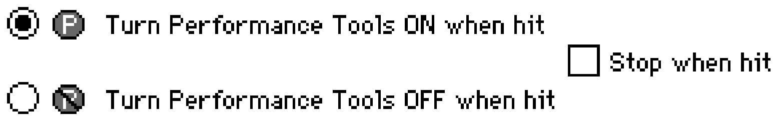
You may want to measure the performance of only certain portions of your code rather than the entire program. One method is to turn performance measurement on and off manually when your program is stopped by choosing Enable Utility or Disable Utility from the Performance submenu.

Another method is to use performance breakpoints, which turn sampling on or off at a given point in the program. By using them, you can create zones of your program where sampling will take place and zones where it will not take place.

To set a performance breakpoint, hold down the Control key and click in the breakpoint column next to a statement in a code window. The Breakpoint Options dialog box appears; for more information, see "The Breakpoint

Options Dialog Box" (page 90). Figure 6-4 shows the part of the dialog box that you use to set performance breakpoints.

Figure 6-4 Selecting a performance breakpoint



● Turn Performance Tools ON when hit

○ Turn Performance Tools OFF when hit

☐ Stop when hit

For each breakpoint, you click one of the radio buttons to specify whether Performance Tools should be turned on or off.

- If you turn Performance tools on, sampling begins and continues until an "Off" performance breakpoint is reached.
- If you turn Performance tools off, sampling stops and stays off until an "On" performance breakpoint is reached.

If you click the "Stop when hit" checkbox, the performance breakpoint acts as a real breakpoint as well; that is, the program stops when the statement is reached. Otherwise, the ASP is turned on or off but the program does not stop.

Note
Take care when using performance breakpoints. Because it takes time for the debugger to process performance breakpoints, inaccuracy can be introduced into the measurement of time-critical paths (for instance, when processing events in the event loop). ◆

You can also enable and disable performance measurement manually by placing simple breakpoints in your code and then choosing Enable Utility or Disable Utility from the Performance submenu when a breakpoint is reached.

Generating a Performance Report

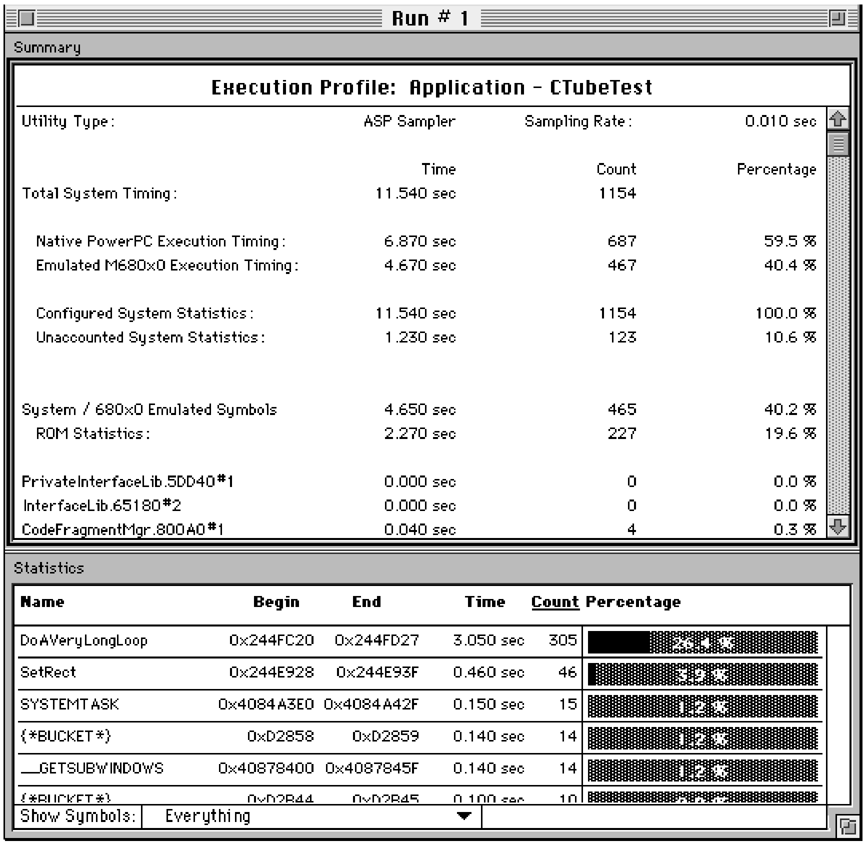
To generate a performance report, you

- stop the program that is being measured
- choose Gather Report from the Performance submenu

The ASP then collects and sorts the performance information and updates the performance window. Figure 6-5 shows a performance window containing data from a report.

The window has two panes: the statistics view and the summary view.

Figure 6-5 A performance window after a report has been gathered



```
Run # 1
Summary
                Execution Profile:  Application - CTubeTest
Utility Type:                              ASP Sampler     Sampling Rate:          0.010 sec

                                                  Time              Count          Percentage
Total System Timing:                        11.540 sec               1154

  Native PowerPC Execution Timing:           6.870 sec                687              59.5 %
  Emulated M680x0 Execution Timing:          4.670 sec                467              40.4 %

  Configured System Statistics:             11.540 sec               1154             100.0 %
  Unaccounted System Statistics:             1.230 sec                123              10.6 %


System / 680x0 Emulated Symbols              4.650 sec                465              40.2 %
  ROM Statistics:                            2.270 sec                227              19.6 %

PrivateInterfaceLib.5DD40#1                  0.000 sec                  0               0.0 %
InterfaceLib.65180#2                         0.000 sec                  0               0.0 %
CodeFragmentMgr.800A0#1                      0.040 sec                  4               0.3 %

Statistics
Name                    Begin         End          Time    Count  Percentage
DoAVeryLongLoop     0x244FC20   0x244FD27     3.050 sec      305  26.4 %
SetRect             0x244E928   0x244E93F     0.460 sec       46  3.9 %
SYSTEMTASK         0x4084A3E0  0x4084A42F     0.150 sec       15  1.2 %
(*BUCKET*)            0xD2858     0xD2859     0.140 sec       14  1.2 %
__GETSUBWINDOWS    0x40878400  0x4087845F     0.140 sec       14  1.2 %
(*BUCKET*)            0xD2B44     0xD2B45     0.100 sec       10
Show Symbols:  Everything
```

The following sections provide more detail on the information in the window.

The Statistics View

The statistics view shows the number of hits registered in each memory range, or bucket. If possible, the name of the routine associated with a bucket is displayed. Each row shows information about a specific routine or an unnamed bucket, showing starting and ending addresses and performance data. The performance data is shown in three ways:

- time (number of hits times the sampling rate)
- count (actual number of hits recorded)
- percentage (number of hits divided by total hits for the session)

The routine names can be identified if

- a bucket is in an address range associated with one of the fragments linked to your application and there's a symbol file for that fragment
- a MacsBug symbol exists whose address falls within the bucket

Otherwise the ranges are identified with the name `{*BUCKET*}`.

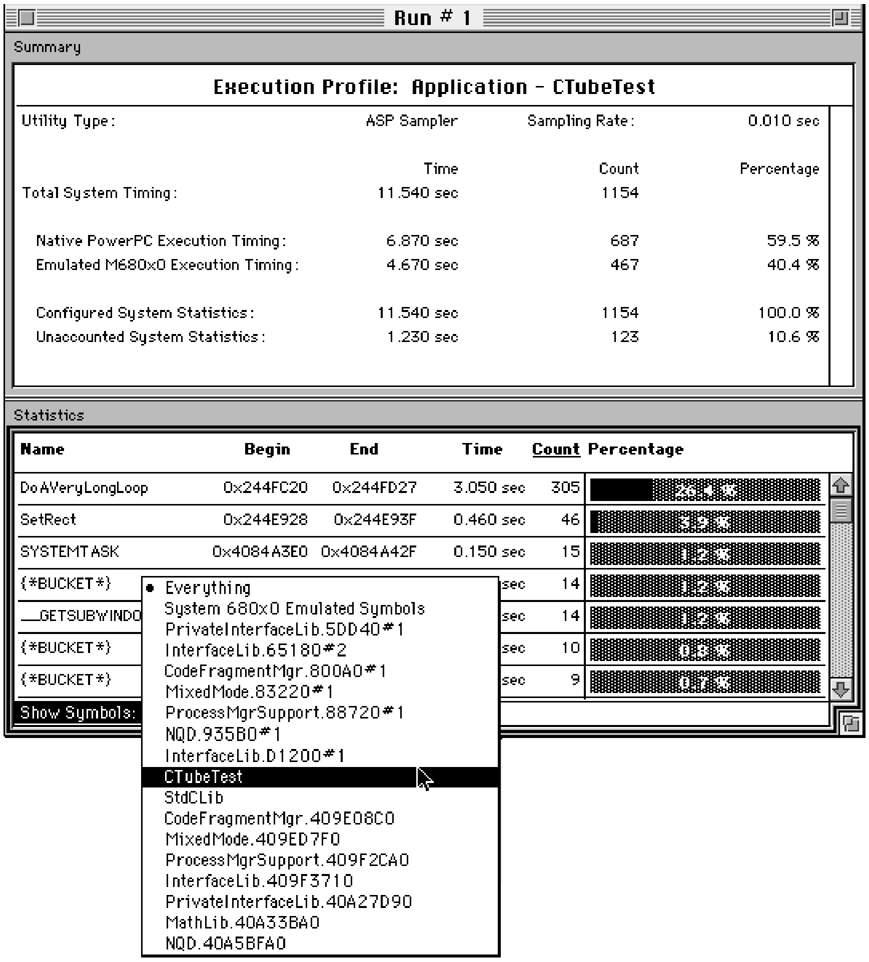
Note
Because function boundaries are unlikely to correspond exactly to bucket boundaries, errors can arise when the ASP associates functions with buckets. The ASP computes the number of hits for a function as the percentage of code the function has in the bucket times the number of hits in the bucket. ◆

By default, the functions and unnamed buckets are listed in descending order by count, that is, by number of hits. You can sort the list alphabetically by name, or by beginning and ending address, by clicking on the appropriate column heading.

If you double-click a routine name, the source code for that routine is displayed in the Browser window. Alternatively, you can select the routine-name and choose Show Source from the Performance submenu. If you double-click an unidentified bucket, the assembly code starting at the beginning address of the bucket is displayed in an instructions window.

By using the Show Symbols pop-up menu in the lower-left corner of the performance window, you can set the fragment or fragments you wish to see hits for. By default, the Show Symbols pop-up menu is set to Everything, meaning that you will see the results for all fragments. In Figure 6-6, the fragment CTubeTest is selected.

Figure 6-6 Filtering data by fragment



Run # 1

Summary

Execution Profile: Application – CTubeTest

Utility Type:	ASP Sampler	Sampling Rate:	0.010 sec
	Time	Count	Percentage
Total System Timing:	11.540 sec	1154	
Native PowerPC Execution Timing:	6.870 sec	687	59.5 %
Emulated M680x0 Execution Timing:	4.670 sec	467	40.4 %
Configured System Statistics:	11.540 sec	1154	100.0 %
Unaccounted System Statistics:	1.230 sec	123	10.6 %

Statistics

Name	Begin	End	Time	Count	Percentage
DoAVeryLongLoop	0x244FC20	0x244FD27	3.050 sec	305	26.4 %
SetRect	0x244E928	0x244E93F	0.460 sec	46	3.9 %
SYSTEMTASK	0x4084A3E0	0x4084A42F	0.150 sec	15	1.2 %
(*BUCKET*)			sec	14	1.2 %
__GETSUBWINDO			sec	14	1.2 %
(*BUCKET*)			sec	10	0.8 %
(*BUCKET*)			sec	9	0.7 %

Show Symbols:

The Summary View

In the summary view, the performance data is grouped into several categories, showing total time, number of hits, and percentage for each. The categories are

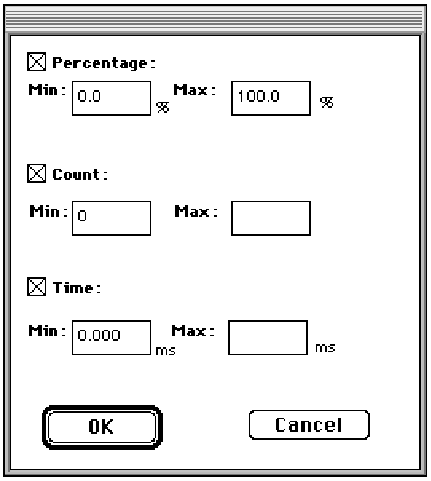
- **Total System Timing.** Total hits recorded for the entire session.
- **Native PowerPC Processor Execution Timing.** Hits recorded in native PowerPC code.
- **Emulated M680x0 Execution Timing.** Hits recorded in emulated 68K code.
- **Configured System Statistics.** Hits mapped to specific fragments.
- **Unaccounted System Statistics.** Hits that cannot be clearly associated with specific fragments.
- **System/680x0 Emulated Symbols.** Hits in code executing outside of any fragment associated with the application being measured. This category includes the ROM and Toolbox code.
- **ROM Statistics.** Hits in an address range in ROM.

Following these categories is a breakdown of statistics by fragment, listed in ascending order by address of the fragment. For an example of such a list, see the bottom of the summary view in Figure 6-5 (page 128).

Editing the Performance Report

At times you may find it useful to display a subset of the data rather than showing statistics for every bucket. If you choose Configure Report from the Performance submenu, the ASP displays a dialog box like the one in Figure 6-7.

Figure 6-7 The report configuration dialog box



☒ Percentage:
Min: 0.0 % Max: 100.0 %

☒ Count:
Min: 0 Max:

☒ Time:
Min: 0.000 ms Max: ms

This dialog box allows you to specify a maximum or minimum value for percentage of hits, number of hits, and time elapsed. You may, for example, want to show only buckets that have at least 1% of the total number of hits. To do so, click the Percentage checkbox, and enter “1.0” in the Min text field. You can also specify a maximum value. Leaving the Max text field blank means there is no upper bound.

You may check more than one checkbox. For example, if you check Percentage and specify a minimum of 13% and also check Count and specify a minimum of 10 hits, only routines that satisfy both conditions are shown.

Saving and Printing Performance Data

Choose Save Window As Text from the File menu to save performance data to a tab-delimited MPW text file. You can view the file in any program that reads text files. You cannot reload and view the file using the Power Mac Debugger.

Note
The saved file contains the performance data in its entirety, regardless of how you have configured the statistics view. ◆

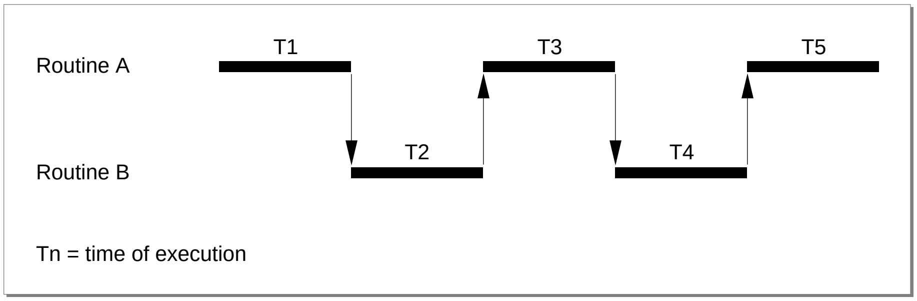
You can print either the summary view or the statistics view of a performance window. To do so, make the view you want the active pane and choose Print from the File Menu.

Evaluating Performance Data

After the ASP collects, sorts, and displays performance data, it's up to you to interpret the data. This section offers some suggestions on interpreting your performance data.

The ASP displays flat-time measurements. **Flat time** is the amount of time spent executing a routine, not counting time spent in called routines. For example, in Figure 6-8, in which routine A calls routine B, the flat time for routine A is equal to T1 + T3 + T5.

Figure 6-8 Flat-time measurement



It's best to generate several performance reports before drawing general conclusions about your application's performance. Any one invocation may suffer from "blind spots," due either to a partial use of the application's code or to measurement inaccuracies. The following section discusses some of the possible sources of inaccuracy in the results.

Possible Problems Or Errors

Because of the statistical nature of sampling, the results you obtain in any run may contain small errors or inaccuracies. There are three major categories of errors:

- **Resonance effects**

  A resonance effect can occur when an application's code has a loop that executes with a regular period. If the frequency of the loop's execution is a multiple of the sampling rate, the ASP will always register hits for the same instruction in that loop. Other instructions will never register, even though they may execute frequently.

  To correct this problem, reset the sampling rate to be slightly lower or higher than its current value and take new measurements.

- **Dynamic splitting and reformation errors**

  You may occasionally see a hit in a memory area you know your program did not execute in. This type of error, which occurs because of the way the ASP adjusts the size of its buckets as it takes samples, tends to be statistically insignificant. The next section, "How the ASP Gathers Data," explains the sampling process further.

- **Overhead**

  There is a certain amount of overhead involved when running your application with the ASP enabled. In other words, the act of measuring performance has an effect on performance.

  Tests have determined that the total overhead is about 3% of the total time. Of that total time, 2% is measured overhead; that is, it is included in the statistics. The other 1% is unmeasured overhead. In other words, if you perform certain tasks with your application and look at the total time *x* reported by the ASP, you can assume the tasks would have taken about 98% of *x* without the ASP running.

How the ASP Gathers Data

This section describes how the ASP registers hits and how it matches hits with distinct memory regions. You do not need to read this section to use the ASP.

However, reading it can help you better understand the data in the performance report.

The ASP registers hits that occur anywhere in memory, not only in your application’s code but also in shared libraries, system software, ROM code, and any other running applications.

Traditional sampling tools allocate a large number of buckets of fixed size. This allocation takes a lot of memory and can result in a poor level of granularity for the regions that execute most frequently. The ASP does not associate a fixed-size region of memory with each bucket. Instead, it dynamically adjusts the sizes of the buckets as it takes the samples. It uses smaller buckets for sections of code that execute most often and larger buckets for infrequently used sections. This allows it to use a relatively small, fixed buffer size.

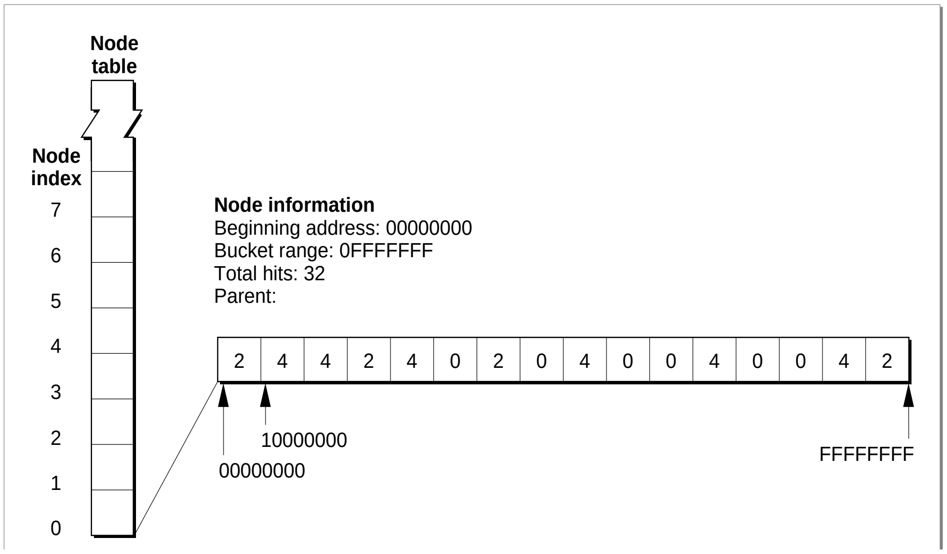
The ASP uses an array called a **node table,** consisting of several nodes. Nodes are divided into 16 buckets of equal size. Each node stores its address range, its total number of hits, and the number of hits in each bucket.

When sampling begins, only one node (node index 0) is active. This node is divided into 16 buckets that cover the entire address space (0x0000 0000 to 0xFFFF FFFF). Each bucket covers has a range of 0x1000 0000 bytes (2^28).

Every time the ASP samples the PC, it determines which bucket the PC belongs in and increments the bucket’s count. Figure 6-9 shows what the first node might look like after 32 samples have been taken.

When a bucket receives more than a certain number of hits (the threshold), a new node is added to the table, corresponding to that bucket, and the bucket is split into 16 new buckets.

Figure 6-9 Recording bucket hits in a node

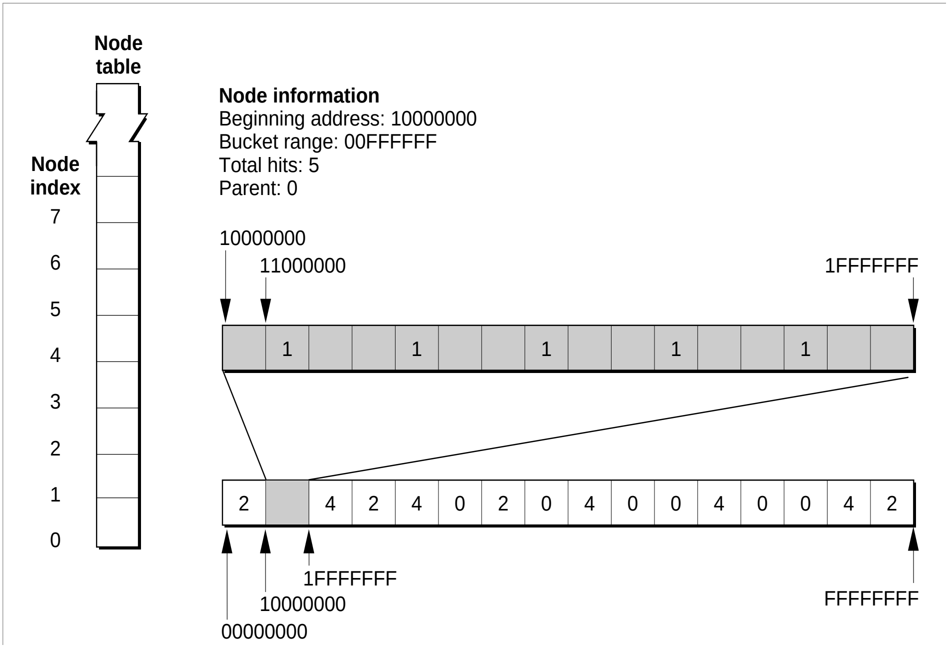


Note
The following formula is used to calculate the threshold: $T = \max(5, (S/128))$ where S is the total number of samples taken. In other words, for a small number of samples the threshold (T) is 5; as the number of samples taken increases, the threshold increases in proportion with the number of samples taken. ◆

In this example, suppose the next sample registers a hit in the second bucket from the left, thus increasing its count to 5. Since 5 is the threshold, the ASP splits the bucket. It allocates a new node, containing 16 buckets, each of which is 1/16 the size of the previous bucket; in this case the new buckets are 2^24 in size.

Figure 6-10 shows what the node table and two allocated nodes look like after the bucket is split.

Figure 6-10 Splitting a bucket



When the ASP splits a bucket, because it no longer has exact PC information for hits in the bucket, it distributes the hits evenly to the buckets in the new node. In this example, the 5 existing hits are shown evenly spread among the new buckets. This distribution can result in some inaccuracy, because any individual hit might be assigned to the wrong bucket. However, this error is usually insignificant, because as the new buckets fill up, the erroneous hits become only a small percentage of the total.

As hits accumulate further, the ASP continues to split buckets as necessary until it cannot split them any more (that is, it reaches a bucket size of 1 byte per bucket). Since ASP uses a fixed-size node table, it's possible that when a bucket needs to be split, there may be no more available entries in the table. To solve this problem, the ASP selects the five least frequently used nodes and frees them from the table, reallocating their hits to the parent node. It then uses the freed nodes for the more frequently used buckets. Although this process causes some loss of information, the error is insignificant because the reformulated buckets have a low density of hits.

CHAPTER 7

Troubleshooting

Contents

CHAPTER 7

Troubleshooting

This chapter provides solutions to the problems that developers most commonly encounter when using the Power Mac Debugger.

Targeting and Mapping

Symbol File Not Mapped

Problem

You opened a symbol file, but the status panel reads “Not mapped”. When you set a breakpoint, it appears as a sad face, like the one shown in Figure 4-7 (page 93).

Solution

First make sure the application you want to debug is running. If it is not, launch it either by choosing Launch from the Control menu or by holding down the Control key while launching it from the Finder. These procedures cause a break on launch and cause the application to be targeted.

If the application was already running when you opened the symbol file, it is not automatically targeted. Instead, you must explicitly target the application by using the Process Browser. See “The Process Browser” (page 73) for instructions.

If the application is targeted but the debugger is unable to map the code to the symbol file, the message “Not mapped” is displayed in the Browser’s status panel. This message can occur for one of two reasons:

- The symbol file name differs from the fragment name, and you have disabled the “Always auto-map symbolics to code” option. As a result, the debugger does not match the fragment to the symbol file by size. For more information, see “Browser Preferences” (page 153).
- The option is enabled but there is more than one fragment of the correct size. As a result, the debugger does not do the mapping.

In either of these cases, you need to do the mapping explicitly by choosing Map Symbol File from the Control menu. See “Mapping Symbols To Code” (page 41).

Can’t Map a Fragment

Problem

The code fragment you want to map does not appear in the Map Symbol File dialog.

Solution

The Map Symbol File dialog box shows only fragments associated with the currently focused process. You may have to change the current focus in order to see your fragment. For example, if you are debugging code that resides in the system heap, you must change the current focus to the system process by double-clicking its name in the Process Browser. This operation is described in “The Current Focus” (page 76).

Using the Serial Nub

Can’t Open Windows

Problem

When you’re using the serial nub and open the Process Browser, no applications (or not all running native applications) are listed. Also, the debugger won’t let you open most of the other windows.

Solution

When using the serial nub, a process must be stopped for any communication to take place between the host and the nub. The information in the Process

Browser (and all other debugger windows) is not updated unless a process is stopped under debugger control.

If you open the Process Browser before having stopped in any process, it does not display any information. The first time you stop a process, the currently running applications are displayed. If this window does not appear to be updating correctly, you may need to open and close the window.

You can stop a process by doing a break on launch, by having `DebugStr` or `Debugger` calls in your program, or by using the Stop command. Once you have stopped the machine, you can use all the features of the debugger to look at the stack and registers, to display and change memory, to set breakpoints and single-step, and so on.

Can’t Debug Threads

Problem

Your program uses the Thread Manager to create cooperative threads, but the Process Browser doesn’t display them.

Solution

The serial nub cannot target individual threads, only processes. You must use the application nub (Power Mac DebugServices) in order to debug individual threads.

Connecting Two Machines

Can’t Connect to the Target Machine

Problem

You are trying to use a two-machine setup, with the application nub on the target machine. You can’t connect to your target machine.

Solution

Here's how to connect to the target machine using AppleTalk:

1. **Make sure both machines are on an AppleTalk network.**
2. **Make sure the application nub (Power Mac DebugServices) is installed and running on the target machine.**
3. **Make sure Program Linking is enabled on the target machine.**

   Open the Sharing Setup control panel on the target machine. Set the names and password if they are not already specified. Turn on Program Linking if it is not already on.
4. **Make sure you are a user on the target machine.**

   Open the Users & Groups control panel on the target machine. If you are not already a user, select New User from the File menu. Open your user icon by double-clicking it, then make sure the Program Linking checkbox is turned on.

▲ **WARNING**
Do not enable program linking for Guests unless you want to give everyone else on the network access to your computer. ▲

5. **When you launch the debugger, select AppleTalk when you are prompted for the connection type.**

   If the dialog box shown in Figure 2-3 (page 38) does not appear, you may have already selected a default connection type. To change the default connection type, choose Connection Preferences from the Edit menu.
6. **Select your target machine.**

A dialog box like the one shown in Figure 2-4 (page 39) prompts you to select the target machine. The list displays only PowerPC-based Mac OS computers currently running the application nub (Power Mac DebugServices). Select your target machine and click OK. A Program Linking dialog box now asks for your user name and password. Enter them and click OK.

Can’t Reconnect to the Target Machine

Problem

Your target application crashed and you relaunched it. Now the debugger does not target the application correctly.

Solution

If the target crashes when you are using the application nub, you must quit the Power Mac Debugger on the host machine and relaunch it.

Source-File Problems

Statement Markers Not Correct

Problem

When you single-step through your source file, the statement markers are not correct. Statements are highlighted partially, or they overlap.

Solution

The symbol file and source file may be out of date with respect to each other. You may have more than one version of the source code on your disk, and the debugger is using a different source file than was used to create the symbol file. To correct this, choose Locate Correct Source File from the File menu. A standard file dialog box appears, allowing you to navigate to the correct source file.

To find out which source file the debugger is using, choose Show Full Path Name from the File menu.

Source Code Not Displayed

Problem

The debugger does not display source code for a routine.

Solution

Symbol information may not be available for that file (because the file was not compiled with the correct compiler option).

Another possibility is that symbol information is available but the debugger cannot locate the source file. In this case, you are prompted with a standard file dialog box to locate the file.

Assembly Code Not Displayed

Problem

The debugger does not display assembly code for a routine.

Solution

Assembly code is available only when the application is targeted. If the application is not running, launch it with the Control key down and it will be automatically targeted. If it is already running, use the Process Browser to target it.

Other

Can’t Single-Step

Problem

You can set a breakpoint in your application, but when you single-step, the program counter arrow disappears and you can’t tell where you are.

Solution

Make sure your application has a `'SIZE'` resource (with resource ID –1), and make sure its `CanBackground` bit is set to `TRUE`.

Global Variables Not Visible

Problem

Not all your program’s global variables are visible in the Global Variables window.

Solution

Some global variables live in “data-only” fragments. To display the values of these variables, you must open a symbol file corresponding to their fragment. Because the fragment contains data only, the Browser window that opens does not display any functions.

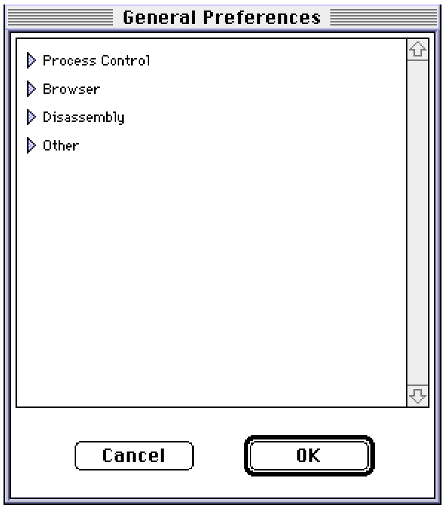
Appendixes

Debugger Preferences

You can alter the Power Mac Debugger's behavior according to your debugging needs. The debugger provides several preferences that you can set.

To set a preference, choose General Preferences from the Edit menu. A dialog box appears, as shown in Figure A-1. In it are four categories of preferences: Process Control, Browser, Disassembly, and Other. To show the preferences under each category, click a disclosure triangle.

Figure A-1 The General Preferences dialog box



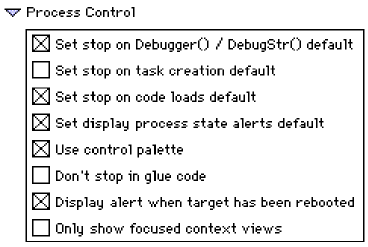
The rest of this appendix describes the preferences for each category.

Note
The preferences you set are stored in the Power Mac Debugger Prefs file, which is located in the Macintosh Debugger Preferences folder in the Preferences folder of the host machine's System Folder. The first time you run the debugger, a default set of preferences is created. ◆

Process Control Preferences

Process control preferences (Figure A-2) include settings that determine whether or not the debugger is entered by default under certain conditions, such as `DebugStr` calls. You can override these settings for any running process or thread by using the Process Browser's preferences; see "Application Preferences" (page 77).

Figure A-2 Process control preferences



▽ Process Control

- ☒ Set stop on Debugger() / DebugStr() default
- ☐ Set stop on task creation default
- ☒ Set stop on code loads default
- ☒ Set display process state alerts default
- ☒ Use control palette
- ☐ Don't stop in glue code
- ☒ Display alert when target has been rebooted
- ☐ Only show focused context views

- **Set stop on Debugger()/DebugStr() default.** If this box is checked, the debugger is entered whenever a `Debugger()` or `DebugStr()` statement is executed in any process.
- **Set stop on task creation default.** If this box is checked, the debugger is entered whenever a process creates a new thread using the Thread Manager.
- **Set stop on code loads default.** If this box is checked, the debugger is entered whenever a targeted process loads any code (for example, when an application loads a code fragment). The debugger displays an alert

indicating which fragment was loaded. This is useful when you want to debug, say, a code resource that your application loads.

This option does not cause a newly launched process to stop when its code fragment is loaded; for that, you must explicitly do a “break on launch.”

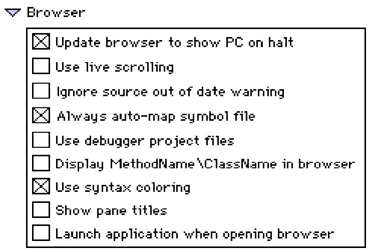
- **Set process state alerts default.** If this box is checked, the debugger displays alerts under certain conditions, such as when a break on launch occurs or when a process quits. If the box is unchecked, the alerts are not displayed.
- **Use control palette.** If this box is checked, the control palette is displayed automatically when you launch the debugger. If the box is unchecked, the control palette is not displayed. You can, however, display it manually.
- **Don’t stop in glue code.** Whenever a PowerPC program makes a system call, a few instructions of **glue code** generated by the compiler are executed before the system call is actually entered. Glue code is also generated, for example, when an application makes cross-fragment calls or C++ method calls. If you step into one of these calls, by default you first step into the glue code. If this box is checked, the debugger skips over the glue code and takes you directly to the first instruction in the called routine.

  System glue code can be accessed from the Browser by selecting the entry called “•••synthesized glue•••” (at the end of the Files pane).
- **Display alert when target has been rebooted.** If this box is checked, when using a two-machine setup, the host displays an alert when it detects that the target machine has been rebooted.
- **Only show focused context views.** If this box is checked, the debugger hides all stack and registers windows associated with contexts other than the current focus. These windows are displayed again when their context becomes the current focus. If the box is unchecked, all windows you have opened are visible.

Browser Preferences

Browser preferences (Figure A-3) include settings that affect the behavior and appearance of the Browser, as well as other debugger windows.

Figure A-3 Browser preferences



Browser

- [✓] Update browser to show PC on halt
- [ ] Use live scrolling
- [ ] Ignore source out of date warning
- [✓] Always auto-map symbol file
- [ ] Use debugger project files
- [ ] Display MethodName\ClassName in browser
- [✓] Use syntax coloring
- [ ] Show pane titles
- [ ] Launch application when opening browser

- **Update browser to show PC on halt.** If this box is checked, the Browser window displays the function containing the program counter whenever a program stops. If the box is unchecked, the Browser continues to display whatever it was already displaying.
- **Use live scrolling.** If this box is unchecked, and you drag the thumb in a the scroll bar of a debugger window, the contents of the window change do not change until the thumb stops moving. If the box is checked, the window's contents change as the thumb moves.
- **Ignore source out of date warning.** If this box is unchecked, the debugger warns you if the symbol file is out of date with respect to the target application or the source files. If you know that any changes do not affect the mapping between files, you can check this option and the alerts will not appear.
- **Always auto-map symbol file**. If this box is checked, if the debugger cannot map a symbol file to a fragment by name (for example, `MyApp.xcoff` with a fragment called `MyApp`), it looks for a fragment whose size is the same as the one specified in the symbol file, and uses it for the mapping. Uncheck this option if you do not want this mapping to occur.
- **Use debugger project files.** If this box is checked, the debugger keeps track of information about your debugging session and can restore its state if you quit and resume later. It stores information about
  - the size and location of the Browser window
  - breakpoints
  - the path to your source files

  The information is stored in files with the suffix `.dbg`. For example, if your symbol file is `MyApp.xcoff`, the project file is `MyApp.dbg` and is stored in the same directory as the symbol file (unless the symbol file is on a read-only volume, in which case you are asked to create an alias to it on a writable volume).

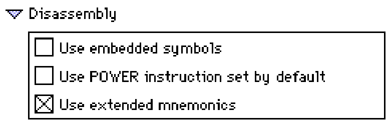
When using the Open command from the File menu, if you select a `.dbg` file, the debugger automatically opens the corresponding symbol file.

- **Display MethodName\ClassName in browser.** If this box is checked, C++ method names are displayed in the form `MyMethod\MyClass` instead of the standard form `MyClass::MyMethod`. Check this option if you want to use keyboard navigation in the Browser's function list. Because source files often contain several methods belonging to the same class, typing the first few letters of the class name is not helpful when the class name is displayed first. By using the alternate notation, you can type the first few letters of the function name to locate it.
- **Use syntax coloring.** If this box is checked, C and C++ code is displayed color-coded: language keywords are shown in blue and comments are shown in red.
- **Show pane titles.** If this box is unchecked, debugger windows that have several panes do *not* display the titles of the panes, in order to save screen space. For example, the Files, Functions, and Code panes in the Browser window appear without titles. If you check this option, the titles appear.
- **Launch application when opening browser.** If this box is checked, whenever you open a symbol file, the debugger launches the corresponding application. This option goes into effect only when you are debugging on a single machine and your application and symbol file are in the same directory.

Disassembly Preferences

Disassembly preferences (Figure A-4) control the appearance of windows displaying assembly-language code.

Figure A-4 Disassembly preferences.



Disassembly

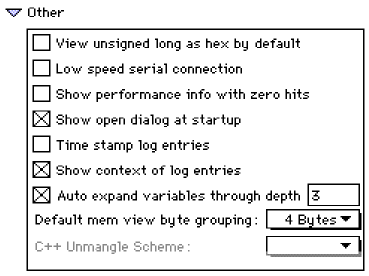
- [ ] Use embedded symbols
- [ ] Use POWER instruction set by default
- [x] Use extended mnemonics

- **Use embedded symbols.** If this box is checked, the debugger displays function name symbols in stack and instructions windows when no symbol file is available. This option is useful if you are doing low-level debugging. If you do not need to use this option do not select it, because it slows the debugger.
- **Use POWER instruction set by default.** If this box is checked, the debugger uses POWER mnemonics in instructions windows. If the box is unchecked, the debugger uses PowerPC mnemonics.
- **Use extended mnemonics.** If this box is checked, the debugger uses extended mnemonics in PowerPC instructions displays. (If you selected the “Use POWER instruction set by default” option, this preference does nothing.)

Other Preferences

Figure A-5 shows the other preferences that you can set.

Figure A-5 Other preferences



Other

- [ ] View unsigned long as hex by default
- [ ] Low speed serial connection
- [ ] Show performance info with zero hits
- [x] Show open dialog at startup
- [ ] Time stamp log entries
- [x] Show context of log entries
- [x] Auto expand variables through depth 3

Default mem view byte grouping: 4 Bytes

C++ Unmangle Scheme:

- **View unsigned long as hex by default.** If this box is checked, unsigned long values are shown in hexadecimal in variable displays. If the box is unchecked, unsigned longs are shown as decimal values.
- **Low speed serial connection.** Check this box if your host machine is slower than a Macintosh Quadra and you experience communication problems between host and target machines. You must also check the low-speed serial connection box in the Debugger Nub Controls control panel. See Figure 1-4 (page 29).
- **Show performance info with zero hits.** If this box is unchecked, the Adaptive Sampling Profiler does not display buckets with zero hits in performance reports. Check this option to include buckets with zero hits.
- **Show open dialog at startup.** If this box is checked, the debugger prompts you to open a symbol file when it launches. If you uncheck this option, the debugger does not prompt you.
- **Time stamp log entries.** If this box is checked, any text that is written to the Log window (such as output from debugger extensions or calls to `DebugStr`) is marked with the date and time.
- **Show context of log entries.** If this box is checked, the Log window displays the application or thread that generated the Log entry, such as with a `DebugStr` call.
- **Auto expand variables through depth *n*.** When displaying variables in evaluation windows, nested variables are automatically expanded through this number of levels; the default is 3. See "Expression Results Windows" (page 105) for more information.
- **Default mem view byte grouping.** By default, memory is displayed (and edited) in groups of 4 bytes. Choose a different value from the pop-up menu to set the default memory display at a 2-byte or 1-byte grouping.
- **C++ Unmangle Scheme.** If you have created a custom C++ unmangle scheme, you can use this option to make it active. Refer to Appendix D, "Creating Custom Unmangle Schemes," for more information on unmangle schemes.

Expression Evaluation

The Power Mac Debugger allows you to use expressions in these situations:

- When setting conditional breakpoints, you specify an expression that must evaluate to `TRUE` for the program to stop at the breakpoint.
- When you choose Evaluate from the Evaluate menu, a dialog box allows you to enter an expression to be evaluated.

The debugger's expression evaluator supports a subset of C and C++ grammar. It allows some additional expression syntax, including the use of register names and other ways of specifying constants.

Additions to C/C++ Syntax

This section shows how you use register names and constants in expressions that the debugger can evaluate.

Register Names

Register names have predefined meaning in the debugger, as shown in Table B-1. However, register names are not reserved words. If you declare a variable to have the same name as a register, the expression evaluator recognizes it as a variable name. If you want to refer to the register by that name in the same program, you must add the delta (∆) symbol before the register name. To do so, type Option-J. For example, if your source code contains the following declaration,

```
Int R1=10;                    /* R1 as a variable */
```

you can evaluate this expression, in which `R1` is a reference to a variable, as

```
x + R1
```

You can also evaluate the following expression, in which `R1` is a reference to register R1:

```
x + ∆R1
```

The same rule applies to references to the floating-point registers and to the special-purpose registers.

Note
To speed up processing, use the ∆ prefix in register names, whether or not you have also declared those names to be variables. ◆

Table B-1 PowerPC register names

Register name	Use
R0–R31	General-purpose registers
FP0–FP31	Floating-point registers
SP	Stack pointer
TOC	Pointer to the Table of Contents (TOC)
PC	Program counter
LR	Link Register
CR	Condition Register
CTR	Count Register
XER	Integer Exception Register
MQ	Register extension for multiply and divide
FPSCR	Floating-Point Status and Control Register

Constants

The expression evaluator handles constants according to the rules defined in ANSI C and *The C Programming Language* by Kernighan and Ritchie. In addition,

- a decimal number (consisting of the characters 0–9) can be preceded by a pound sign (#)
- a hexadecimal number (consisting of the characters 0–9 and A–F) can be preceded by a dollar sign ($)

A number not preceded by any special characters is treated as hexadecimal if it contains the characters A-F and decimal otherwise. However, keep in mind that a hexadecimal constant such as F9 could cause a conflict with a variable of the same name declared in your program.

What Isn't Supported

The evaluator does not support

- assignments
- member pointers
- type casting
- expressions of the form `sizeof(`*typeName*`)`, although it can handle expressions of the form `sizeof(`*variableName*`)`
- taking the address of a function

Although the evaluator does not support type casting, you can change the type of an expression result by selecting it and choosing Evaluate from the Evaluate menu. See "Expression Results Windows" (page 105).

Debugger Extensions

You can customize and extend the Power Mac Debugger by creating debugger extensions. A **debugger extension** is a C or C++ function that you can execute to perform a task while using the debugger. Typically, developers use debugger extensions to display system data structures such as those that are used to track installed drivers, volumes, open files, and VBL tasks.

This chapter describes how to execute, write and build debugger extensions. It also documents the callback routines you can use when writing debugger extensions.

Executing Debugger Extensions

You execute debugger extensions from the debugger's Log window.

- To get a list of available debugger extensions, enter `extensions` in the Log window.
- To get information about what an extension does, and its required parameters, enter `help` *extensionname* in the Log window.
- To execute an extension, enter its name followed by any necessary parameters in the Log window. All output from the extension is displayed in the Log window.

For more information, see “The Log Window” (page 66).

Writing Debugger Extensions

Debugger extensions contain native PowerPC code and are written in C or C++. After compiling an extension and linking it with the appropriate libraries, you store the executable code in a resource of type `'ndcd'`. (It is similar to a MacsBug extension, which is stored in a code resource of type `'dcmd'`.) Then you add the code resource to the resource fork of the debugger's preferences file, located in the Preferences folder of the target machine's System folder. The debugger loads the extensions into the system heap during system startup.

A debugger extension must contain a function declared as follows:

```
pascal void CommandEntry (dcmdBlock* paramPtr);
```

When the user executes the extension, the debugger calls the `CommandEntry` function and passes it a single parameter, which is a pointer to a parameter block. For more information on this parameter block, see "Data Structures" (page 169). The `request` field of the parameter block can be one of three constants, as shown in Table C-1.

Table C-1 Values passed in the `request` field

Constant	Description
`dcmdInit`	This value is passed when the debugger first calls the extension, after the debugger is loaded into memory. The extension can perform any necessary initialization actions, for example, initializing its global variables or gathering information about the operating environment.
`dcmdDoIt`	This value is passed when the user executes the extension.
`dcmdHelp`	This value is passed when the user asks for information about the extension by typing `help` in the Log Window.

Listing C-1 shows the format of the source code for a debugger extension. Although a debugger extension can contain more than one procedure, the main procedure must be called `CommandEntry`.

Listing C-1 Skeleton code for a debugger extension

```
#include <Types.h>
#include "dcmd.h"
#include "ndcd.h"

/*Declare global variables, if any.*/
pascal void CommandEntry (dcmdBlock* paramPtr){
    /*Declare local variables, if any.*/
    /*The following case statement dispatches on the value
     of the request field of the parameter block.*/
    switch (paramPtr->request){
        case dcmdInit:
            /*code that executes at initialization time*/
        case dcmdHelp:
            /*code that displays syntax and help info*/
        case dcmdDoIt:
            /*code that performs command’s normal function*/
        }
}
```

A Sample Debugger Extension

Listing C-2 shows sample source code for the Echo debugger extension. This command echoes extension parameters; if the parameter is an expression, it evaluates the expression. The sample code makes use of debugger callback routines, which are introduced in the next section, “Using Callback Routines.”

Listing C-2 Sample source code for debugger extension

```
/*File: Echo.c, sample debugger extension*/
#include <Types.h>
#include "dcmd.h"

void NumberToHex (long number, Str255 hex){
    Str255      digits = "0123456789ABCDEF";
    int         n;
    strcpy (hex, ".00000000");
    for (n = 8; n >= 1; n--){
        hex[n] = digits[number % 16];
        number = number / 16;
        }
    }

pascal void CommandEntry (dcmdBlock* paramPtr){
    short       pos, ch;
    long        value;
    Boolean     ok;
    Str255      str;
    switch (paramPtr->request){
        case dcmdInit:
        case dcmdHelp:
            break;

        case dcmdDoIt:
            do {
                /*save position so we can rewind if error*/
                pos = dcmdGetPosition();
                ch = dcmdGetNextExpression(&value, &ok);
                if(ok){
                    /*the expression was parsed correctly*/
                    NumberToHex (value, str);
                else{
                    /*the expression contained an error
                     go back to saved position*/
                    dcmdSetPosition(pos);
                    /*and get it as string*/
                    ch = dcmdGetNextParameter(str);
                    dcmdDrawLine(str);
                }
            while (ch != CR);
            break;
        }
}
```

Using Callback Routines

There are a number of **callback routines** you can use to assist you in writing debugger extensions. Most of these routines are also used by writers of MacsBug `'dcmd'` debugger extensions (hence the prefix `'dcmd'` in the routine names) and are declared in the file `dcmd.h`. In addition, the routines `dcmdReadRegister` and `dcmdWriteRegister` are declared in the file `ndcd.h`. The routines are implemented in the library file `NubExt.xcoff`.

You can use debugger callback routines to

- parse user input
- display help text or other output
- obtain or change the values of PowerPC registers

Building a Debugger Extension

To build and use a debugger extension, you must complete the following steps:

1. **Compile the source code for the extension.**
2. **Link the resulting object file with** `ndcdGlue.o` **and the** `NativeNub` **library.**
3. **Create an** `'ndcd'` **code type resource, using Rez.**
4. **Add the resource to the resource fork of the preferences file.**

   To do so, use Rez, or a resource editor such as ResEdit.

   If you are using the serial nub, add the resource to the Debugger Nub Preferences file. If you are using the application nub, add the resource to the Power Mac DebugServices Prefs file. Both of these files are in the Preferences folder of the target machine's System Folder.
5. **Reboot the machine.**

   The new debugger extension will be available to execute after you reboot.

Table C-2 lists the files you need to build a debugger extension.

Table C-2 Header and library files for debugger extensions

File	Use
`ndcd.h`	Header file containing declarations for `dcmdReadRegister` and `dcmdWriteRegister` routines
`dcmd.h`	Header file declaring all other callback routines (also used by MacsBug `'dcmd'` extensions)
`ndcdGlue.o`	Standard glue file that you must link with
`NativeNub`	Library file containing the implementation of the routines declared in `dcmd.h` and `ndcd.h`
`Put.c`	A set of formatting routines that can be used by debugger extensions
`Put.h`	Header file for the formatting routines

Debugger Extension Reference

This section describes the constants, data structures, and routines that you use in writing a debugger extension.

Constants

The following constants are used to define values returned in the `request` field of the parameter block passed back to the extension by the debugger.

```
#define dcmdInit 0
#define dcmdDoIt 1
#define dcmdHelp 2
```

The following constants are used to specify register names for the `dcmdWriteRegister` and `dcmdReadRegister` functions:

```
#define R0Register         0
#define R1Register         1
#define R2Register         2
#define R3Register         3
#define R4Register         4
#define R5Register         5
#define R6Register         6
#define R7Register         7
#define R8Register         8
#define R9Register         9
#define R10Register       10
#define R11Register       11
#define R12Register       12
#define R13Register       13
#define R14Register       14
#define R15Register       15
#define R16Register       16
#define R17Register       17
#define R18Register       18
#define R19Register       19
#define R20Register       20
#define R21Register       21
#define R22Register       22
```

```
#define R23Register       23
#define R24Register       24
#define R25Register       25
#define R26Register       26
#define R27Register       27
#define R28Register       28
#define R29Register       29
#define R30Register       30
#define R31Register       31

#define PCRegister        32
#define LRRegister        33
#define CRRegister        34
#define CTRRegister       35
```

Data Structures

The debugger passes a pointer to the `extensionBlock` data structure when it calls the debugger extension. The data structure is declared as follows:

```
typedef struct {
    long       *registerFile;     /*not used*/
    short      request;           /*initialize;execute; display help*/
    Boolean    aborted;           /*not used*/
} extensionBlock;
```

Field descriptions

`registerFile`	Not used; retained for compatibility with MacsBug extensions, which use the field as a pointer to an array containing the contents of the 680x0 registers. To read or change the value of PowerPC registers, use the callback routines `dcmdReadRegister` and `dcmdWriteRegister`, described in "Utility Routines" (page 173).
`request`	A field with a value of `dcmdInit`, `dcmdDoIt`, or `dcmdHelp`. In response to `dcmdInit`, the extension should perform any required initialization. In response to `dcmdDoIt`, the extension should execute its main function. In response to `dcmdHelp`, the extension should display a help message.
`aborted`	Not used; retained for compatibility with MacsBug extensions.

Callback Routines

This section describes the three types of routines you can call when writing a debugger extension: input routines for parsing user input, output routines for displaying help messages and other output, and utility routines for displaying or changing the PowerPC registers.

Note
The routines `dcmdGetBreakMessage`, `dcmdGetNameAndOffset`, `dcmdGetMacroName`, `dcmdSwapWorlds`, `dmcdSwapScreens`, `dcmdScroll`, and `dcmdDrawPrompt` are also declared in the header file `dcmd.h`. They are stub routines included for compatibility with MacsBug extensions. ◆

Input Routines

You can use the following routines to parse user input.

dcmdGetPosition

Returns a short integer specifying the current command line position.

```
pascal short dcmdGetPosition();
```

dcmdSetPosition

Sets the current command line position.

```
pascal void dcmdSetPosition(short pos);
```

`pos`	The position to set. This should be a value returned by the `dcmdGetPosition` routine.

dcmdGetNextChar

Returns the next character or a return character if the entire line has been scanned.

```
pascal short dcmdGetNextChar();
```

dcmdPeekAtNextChar

Returns the next character on the command line or a return character if the entire line has been scanned. The current command line position is not changed.

```
pascal short dcmdPeekAtNextChar();
```

dcmdGetNextParameter

Copies all characters from the command line to the parameter string until a delimiter is found or the end of the command line is reached. A delimiter can be a space, a comma, or a return character. Both single- and double-quoted strings are allowed on the command line; however, the leading and trailing quotes must be of the same type. The routine returns the ASCII value of the delimiter after the expression.

```
pascal short dcmdGetNextParameter(Str255 str);
```

`str`	The parameter string, stripped of quotes.

dcmdGetNextExpression

Parses the command line for the next expression. All expressions are evaluated to 32 bits. A delimiter can be a space, a comma, or a return character. A space is not treated as a delimiter if it occurs in the middle of an expression. For example, the expression 1 + 2 is evaluated to 3, and the delimiter will be the character following the 2. The routine returns the ASCII value of the delimiter after the expression.

```
pascal short dcmdGetNextExpression(long* value, Boolean* ok);
```

`value`	On exit, the evaluated value of the expression.
`ok`	On exit, if `true`, indicates that the expression was parsed successfully.

Output Routines

You can use the following routines to display help text or to format program output. All output produced by debugger extensions is written to the Log window.

dcmdDrawLine

Draws the text in the Pascal string as one or more lines separated by carriage returns.

```
pascal void dcmdDrawLine(const Str255 str);
```

`str`	The text to be drawn.

dcmdDrawString

Draws the text in the Pascal string as a continuation of the current line.

```
pascal void dcmdDrawString(const Str255 str);
```

`str`	The text to be drawn.

dcmdDrawText

Draws a number of characters starting from the specified pointer, as a continuation of the current line.

```
pascal void dcmdDrawText(StrPtr text, short length);
```

`text`	The beginning address of the characters to be drawn.
`length`	The number of characters to be drawn.

Utility Routines

You can use these routines to obtain or change the value of a PowerPC register.

dcmdReadRegister

Returns the value of the specified PowerPC register.

```
pascal void dcmdReadRegister(short regNum, short regSize,
                              void *regValue);
```

`regNum`	An integer from 0 to 31 that specifies the register number whose contents you want returned. Use integers 32 through 35 to specify special registers. See “Constants” (page 168) for additional information.
`regSize`	An integer specifying the size of the register: 32 specifies a general-purpose register; 64 specifies a floating-point register.
`regValue`	On exit, the value in the register specified by `regNum`.

dcmdWriteRegister

Writes a value to a PowerPC register.

```
pascal void dcmdWriteRegister(short regNum, short regSize,
                        void *regValue);
```

`regNum`	An integer from 0 to 31 that is the register you want to write to. Use integers 32 through 35 to specify special registers. See "Constants" (page 168) for more information.
`regSize`	An integer specifying the size of the register: 32 specifies a general-purpose register; 64 specifies a floating-point register.
`regValue`	The value you want to place in the register.

Creating Custom Unmangle Schemes

C++ code requires an “unmangle” scheme to correctly display routine names.

The Power Mac Debugger is able to load any unmangle scheme as a code resource and use its algorithm to unmangle symbols within the symbol file. This capability lets the user use multiple compilers with varying unmangle schemes with the Power Mac Debugger.

This appendix outlines the steps necessary to package your unmangle scheme as a stand-alone code resource containing either PowerPC or 680x0 code. If you are running the Power Mac Debugger host on a 680x0-based system, you must use a 680x0 code resource. On a PowerPC-based system, you can use either kind.

Here’s the function definition for an unmangle scheme:

```
long unmangle(char *dst, char *src, int limit);
```

The routine returns the number of characters copied to the unmangled string.

Creating a 680x0 Code Resource

1. **If you use global variables in your unmangle scheme, you must set up an A5 world.**

   Listing D-1 shows how to structure your code to set up an A5 world. Listing D-2 shows the code for the routines referenced in Listing D-1. This code is assumed to be in a file called `SAGlobals.c`. The header file for these routines, `SAGlobals.h`, is shown in Listing D-3.

2. **Build the resource.**

   A sample makefile is shown in Listing D-4.

3. **Place the created resource in the Macintosh Debugger Preferences folder.**

4. **Choose General Preferences from the debugger’s Edit menu**

5. Select your scheme as the chosen unmangle scheme from the popup in the "Other" category.

Listing D-1 Setting up an A5 world

```
A5RefTypeA5Ref;
long    oldA5;
long    theReturnValue = 1;

MakeA5World(&A5Ref);
oldA5 = SetA5World(A5Ref);
     .
     .   //your code goes here
     .
SetA5(oldA5);
DisposeA5World(A5Ref);
```

Listing D-2 The file `SAGlobals.c`

```
#include <Memory.h>
#include <OSUtils.h>
#include <SAGlobals.h>
#define kAppParmsSize 32

long A5Size(void);      /* prototype for routine in Runtime.o */
void A5Init(Ptr myA5); /* prototype for routine in Runtime.o */

pascal void MakeA5World(A5RefType *A5Ref)
{
    *A5Ref = NewPtr(A5Size());
    if ((long)*A5Ref)
        A5Init((Ptr)( (long)*A5Ref + A5Size() - kAppParmsSize));
}

pascal long SetA5World(A5RefType A5Ref)
{
    return SetA5( (long)A5Ref + A5Size() - kAppParmsSize);
}
```

```
pascal void DisposeA5World (A5RefType A5Ref)
{
    DisposPtr((Ptr)A5Ref);
}
```

Listing D-3 The file `SAGlobals.h`

```
#include <Types.h>

typedef Ptr A5RefType;

/* MakeA5World allocates space for an A5 world based on the
size of the global variables defined by the module and its
units. If sufficient space is not available, MakeA5World
returns NIL for A5Ref and further initialization is aborted. */

pascal void MakeA5World(A5RefType *A5Ref);

/* SetA5World locks down a previously allocated handle containing
an A5 world and sets the A5 register appropriately. The return
value is the old value of A5. The client should save it for
use by RestoreA5World. */

pascal long SetA5World(A5RefType A5Ref);

/* DisposeA5World simply disposes of the A5 world handle. */

pascal void DisposeA5World(A5RefType A5Ref);
```

Listing D-4 A makefile for a 680x0 unmangle code resource

```
#   File:       MyUnmangle.make
#   Target:     MyUnmangle
#   Sources:    SAGlobals.c
#               unmangle.c

# VARIABLE DEFINITIONS
```

```
App         =   MyUnmangle
MAKEFILE    =   {App}.make
COMPILE     =   SC
Objs        =   unmangle.o      ∂
                SAGlobals.o     ∂
                "{CLibraries}"StdClib.o∂
                "{Libraries}"MacRuntime.o

# DEPENDENCIES

{App} ƒƒ {MAKEFILE}  {Objs}
    Link -rt CUST=1000 -t CODE -c '????' -m unmangle -sg unmangle ∂
        {Objs} ∂
        -o {App}

unmangle.o ƒƒ {MAKEFILE} unmangle.c
    {COMPILE} unmangle.c -o unmangle.o
SAGlobals.o ƒƒ {MAKEFILE} SAGlobals.c
    {COMPILE}  SAGlobals.c -o SAGlobals.o
```

Creating a PowerPC Code Resource

1. **Build the resource.**

   Listing D-5 shows a sample makefile for PowerPC code. Listing D-6 and Listing D-7 show the files used by the Rez commands: the file `MyUnmangle.r1` defines the code resource's ID and type, and the file `MyUnmangle.r2` gives the code fragment a routine descriptor that in turn gives the debugger directions to the new unmangle scheme.

2. **Place the created resource in the Macintosh Debugger Preferences folder.**
3. **Choose General Preferences from the debugger's Edit menu.**
4. **Select your scheme as the chosen unmangle scheme from the popup in the "Other" category.**

Listing D-5 A makefile for a PowerPC unmangle code resource

```
#   File:       MyUnmangle.make
#   Target:     MyUnmangle
#   Sources:    unmangle.c

# VARIABLE DEFINITIONS
App         =   MyUnmangle
MAKEFILE=   {App}.make
COMPILE =   MrC
Objs        =   unmangle.o      ∂
                "{SharedLibraries}"InterfaceLib∂
                "{SharedLibraries}"StdCLib

# DEPENDENCIES
{App}    ƒƒ{Objs} {App}.r1 {App}.r2 {MAKEFILE}
    PPCLink -packdata off -term none {Objs} -o {App} -main unmangle
    Rez {App}.r1 -o {App} -c RSED -t 'CODE'
    Rez {App}.r2 -a -m -o {App} -c RSED -t 'CODE'
    Setfile {App} -a B
    unmangle.o ƒƒ {MAKEFILE} unmangle.c
        {COMPILE} unmangle.c -o unmangle.o
```

Listing D-6 The file `MyUnmangle.r1`

```
        Read 'CUST' (1000) "MyUnmangle";
```

Listing D-7 The file `MyUnmangle.r2`

```
#include "MixedMode.r"

type 'CUST' as 'rdes';

resource 'CUST' (1000)
{
    $00000FF1,      /* this is the Mixed ModeManager procInfo value */
    $$Resource ("MyUnmangle", 'CUST', 1000)
};
```

A P P E N D I X E

Quick Reference Guide

This appendix provides two tables that can help you get started without having to read the rest of the book.

Table E-1 is a guide to performing some of the most common debugging tasks when using the Power Mac Debugger. Table E-2 shows a list of the debugger's windows, summarizes their main functions, and describes how to open them.

APPENDIX E

Quick Reference Guide

Table E-1 Common debugging tasks

Task	How to do it
<u>Getting started</u>	
Open symbol file to begin debugging	Double-click symbol file's Finder icon
	Press Command-O (Open from File menu) in debugger
Break on launch (stop application under debugger control before `main`)	Hold down Control key while launching application
	Select Launch from Control menu or drag application icon onto Power Mac Debugger icon in Finder (single-machine debugging only)
Target a running process for debugging	Press Command-Y (Show Process Browser from Window menu) to display Process Browser; double-click name of process
<u>Displaying Code</u>	
Display source code for a function	Click name of function in Functions pane in Browser window
	Double-click function name in Stack Crawl
Create a "clone" of a code or variable display	Press Option key, then click in pane and drag (applies to any code view, Globals Watch Variables list or Stack Crawl variables)
Disassemble PowerPC code starting at a given address	Select an address in any window and press Command-D (New Instructions Window from View menu)
	Double-click PC address in Stack Crawl
Disassemble 68K code starting at a given address	Select an address in any window and press Command-8 (New 68K Instructions Window from View menu)
<u>Setting breakpoints</u>	
Set simple breakpoint	Click diamond next to statement or instruction in Browser, instructions window, or other code view
Set one-shot breakpoint	Press Option and click next to statement or instruction
Set focused simple breakpoint	Press Command and click next to statement or instruction

Task	How to do it
Set conditional, counting or performance breakpoints	Press Control and click next to statement or instruction (brings up Breakpoint Options dialog box)
Set focused conditional, counting or performance breakpoints	Press Command-Control and click next to statement or instruction (brings up Focused Breakpoint Options dialog box)
Examine currently set breakpoints	Press Command-N (Breakpoint List from Window menu)
Clear breakpoint	Click existing breakpoint icon
	Select breakpoint in Breakpoint List and press Delete
Clear all breakpoints	Select Clear All Breakpoints from Control menu
<u>Program Control</u>	
Step one line of code	Press Command-S (Step Over from Control menu or control palette)
Step into a function	Press Command-T (Step Into from Control menu or control palette)
Step out of a function	Press Command-U (Step Out from Control menu or control palette)
Continue executing a stopped program	Press Command-R (Run from Control menu or control palette)
Change program counter	Click program counter and drag it to another instruction
<u>Examining Memory and Variables</u>	
Examine PowerPC registers	Press Command-K (Registers from View menu)
Display (or edit) memory starting at a given location	Select an address in any window and press Command-M (New Memory Window from View menu)
	Double-click frame address in Stack Crawl
Find a hexadecimal or ASCII value in memory	Press Command-M to display a Memory window; click Search button and enter a value to search for
Examine global variables	Press Command-L (Show Global Variables from Window menu)
Examine local variables	Press Command-J (Stack Crawl from View menu);
Evaluate any expression	Select Evaluate from Evaluate menu
	Select name of variable and press Command-E

Table E-2 The Power Mac Debugger windows

Window	Menu command and key equivalent	Main Functions
Browser	Open (File menu) Command-O	Display source code Set breakpoints See currently executing statement
PowerPC Instructions	New Instructions Window (View menu) Command-D	Disassemble memory to PowerPC code
68K Instructions	New 68K Instructions Window (View menu) Command-8	Disassemble memory to 68K code
Control palette	Show Control Palette (Window menu)	Control execution of program (stop, run, step into, step out, step over, turn continuous step on)
Stack Crawl	Stack Crawl (View menu) Command-J	Navigate call chain Examine local variables
Log	Show Log Window (Window menu)	See output from `DebugStr` calls and debugger extensions
Memory	New Memory Window (View menu) Command-M	Display and edit memory Search for values in memory
General-purpose registers	Registers (View menu) Command-K	Display and edit PowerPC general-purpose registers
Floating-point registers	FPU Registers (View menu)	Display and edit PowerPC floating-point registers
Process Browser	Show Process Browser (Window menu) Command-Y	Display and target running processes and threads Set preferences for entering debugger on a process or thread basis
Fragment Info	Show Fragment Info (Window menu)	Display information about all fragments

Table E-2 The Power Mac Debugger windows

Window	Menu command and key equivalent	Main Functions
Global Variables	Show Global Variables (Window menu) Command-L	Display global variables for all targeted processes
Breakpoint List	Show Breakpoint List (Window menu) Command-N	Display all currently set breakpoints
User Stack Crawl	User Stack Crawl (View menu)	Display stack crawl starting at a user-specified location
Expression Results	Evaluate (Evaluate menu) Command-E	Display value of any variable or expression

Glossary

680x0 Any member of the Motorola 68000 family of microprocessors.

680x0 application An application that contains code only for a 680x0 microprocessor. See also **PowerPC application**.

680x0-based Mac OS computer Any computer containing a 680x0 central processing unit that runs Mac OS system software. See also **PowerPC-based Mac OS computer.**

68K Instructions windows Debugger windows that display a disassembly of 680x0 code starting at a specified address.

Adaptive Sampling Profiler (ASP) A sampling utility that allows you to measure the time spent executing sections of your code.

application nub A debugger nub that allows other applications to run when a process is stopped. Can be used with one- or two-machine debugging.

assembly-level debugging Debugging without the use of a symbol file. You can examine memory and registers and disassemble any area of memory to machine instructions.

break on launch The process of launching an application and stopping it before its `main` function is executed.

breakpoint A location in a program at which execution will stop and control will return to the debugger.

Breakpoint List window A debugger window that displays a list of all currently set breakpoints.

Browser window A three-pane window displayed when you open a symbol file. You can use this window to view the source code for selected functions in your target program.

bucket In performance measurement, a range of memory for which performance samples are taken.

callback routines Routines that you can call from debugger extensions. You use these routines to get user input, to display output, and to get or change the value of PowerPC registers.

call chain A list of routines displayed in the Stack Crawl window, showing the current routine and its callers in sequence.

code fragment See **fragment.**

Code Fragment Manager The part of the Mac OS system software that loads fragments into memory and prepares them for execution. See also **fragment.**

conditional breakpoint A breakpoint that halts program execution when it is encountered and a previously specified condition is true.

control palette A floating window that displays information about the target process and allows you to control its execution.

counting breakpoint A breakpoint that halts program execution after it has been encountered a specified number of times.

current focus The process or thread that is capable of being single-stepped. Only one thread at a time can have the current focus.

debugger extension Code of type `'ndcd'`, used to extend or customize debugger functionality.

debugger nub See **nub**.

exception An error or other special condition detected by the microprocessor in the course of program execution.

exception handler Any routine that handles exceptions.

Extended Common Object File Format (XCOFF) A format of executable file generated by some PowerPC compilers. See also **Preferred Executable Format.**

flat time In performance measurement, the amount of time spent executing a routine, not including the time spent in any called routine.

Floating-Point Status and Control Register (FPSCR) A 32-bit PowerPC register used to store the floating-point environment.

focused breakpoint A breakpoint that is set for a particular context (such as a cooperative thread or a client of a shared library). Execution stops at the breakpoint only when the context for which it was set is executing.

FPU Registers window A debugger window displaying the current value of the PowerPC floating-point registers.

fragment Any block of executable PowerPC code and its associated data.

Fragment Info window A debugger window that displays a list of all fragments on the target machine.

global variable A variable that is accessible by all code in a given program.

Global Variables window A debugger window displaying a list of global and static variables for all currently open symbol files.

glue code Code inserted by the development environment that provides linkage between your code and certain called routines, such as system calls and C++ method calls.

hexadecimal An adjective that describes base-16 notation.

host The debugger's user interface. It presents information transmitted by the nub to the user. See also **nub.**

instructions windows Debugger windows that display a disassembly of PowerPC code starting at a given address.

Link Register (LR) A PowerPC register that holds the return address of the currently executing routine**.**

local variable A variable whose scope and life span correspond to the execution of a specific subroutine.

Log window A debugger window that displays the output of `DebugStr` calls and debugger extensions.

MacsBug An assembly-level debugger for Mac OS.

mapping The correspondence between information in a symbol file and code running on the target machine.

memory windows Debugger windows that display the contents of memory starting at a given address.

Mixed Mode Manager The part of the system software that manages the mixed-mode architecture of Power PC-based Mac OS computers running 680x0-based code (including system software, applications, and stand-alone code modules).

node table The name of an array used by the ASP to track the location of PC samples.

nub Debugger software that resides on the target machine and provides information about the state of the target machine to the host software.

one-shot breakpoint A breakpoint that is in effect once only; execution resumes immediately when you set the breakpoint.

parameter a value passed to a subroutine.

PEF See Preferred Executable Format.

performance breakpoint A breakpoint that starts or stops performance measurement for selected blocks of code.

performance window A window in which the ASP displays the performance data for a single performance session.

Power Mac Debugger An application that allows you to debug PowerPC software at the source-code and assembly-language level.

PowerPC-based Mac OS computer Any computer containing a PowerPC central processing unit that runs Mac OS system software. See also **680x0-based Mac OS computer.**

PowerPC Any member of the family of PowerPC microprocessors. The MPC601 processor is the first PowerPC CPU.

PowerPC application An application that contains code only for a PowerPC microprocessor. See also **680x0 application.**

Preferred Executable Format (PEF) The format of executable files used for PowerPC applications and other software running on Mac OS computers. See also **Extended Common Object File Format (XCOFF).**

process A running application.

Process Browser A debugger window displaying a list of all currently executing native PowerPC processes and threads on the target machine.

program counter (PC) A register in the CPU that stores the location of the next instruction to be executed.

Registers window A debugger window displaying the current values of the PowerPC general-purpose registers.

ROM map A file containing information about symbols in ROM code.

RTOC See **Table of Contents Register.**

sampling rate The frequency with which a PC address is taken in measuring program performance.

serial nub A debugger nub that stops the target machine when an exception occurs. Requires a second machine, connected through a serial port, to run the host.

shared library A code fragment of type `'shlb'` containing routines that can be called by one or more other fragments.

simple breakpoint A breakpoint that stops program execution every time it is encountered.

single-step To execute a program one source-code statement or assembly-language instruction at a time.

SOM See **System Object Model (SOM).**

source-level debugging Debugging with the ability to step through source code and examine variables; made possible by the use of symbol files.

Stack Crawl window A debugger window displaying information about the calling chain of routines on the stack, including local variables.

stack frame The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

symbol A name for a discrete element of code or data in a fragment.

symbolic information file (symbol file) A file produced at link time containing information that allows the debugger to associate source code and variable names with machine instructions and memory locations.

System Object Model (SOM) IBM's System Object Model. Allows programs to make use of objects created in different programming languages.

Table of Contents (TOC) An area of static data in a fragment that contains a pointer to each routine or data item that is imported from some other fragment, as well as pointers to the fragment's own static data.

Table of Contents Register (RTOC) A processor register that points to the Table of Contents of the fragment containing the code currently being executed. On the PowerPC processor, general-purpose register 2 is dedicated to serve as the RTOC.

target The code being debugged or measured by the Power Mac Debugger. Also, the machine that the target code runs on.

targeting The process of informing the debugger that you want to debug a particular application.

thread A cooperative execution context as implemented by the Thread Manager. Threads can be debugged individually using the application nub.

TOC See **table of contents.**

transition vector An area of static data in a fragment that describes the entry point and TOC address of a routine.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Ron Karr

EDITOR
Jeanne Woodward

ILLUSTRATOR
Deborah Dennis

PRODUCTION EDITORS
Alexandra Solinski, Lisa Le

Special thanks to Gary Kratzer

Acknowledgments to Dave Bice, Paul Forrester, Robert Goosey, Sandeep Gupta, Liz Ghini Locklear, Kevin Looney, Herb Ruth, Julie Wang, Mike Wimble