
Macintosh Debugger Reference

Second Edition

For 680x0-based Macintosh and Power Macintosh computers

Apple Computer, Inc.
© 1995 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc.

The Apple logo is a trademark of Apple Computer, Inc.
Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, AppleLink, AppleTalk, Macintosh, Macintosh Quadra, and MPW are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Mac and Power Macintosh are trademarks of Apple Computer, Inc.

Adobe Illustrator and Adobe Photoshop are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

America Online is a registered service mark of America Online, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Motorola is a registered trademark of Motorola Corporation.

PowerPC is a trademark of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD “AS IS,” AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings ix

Preface

About This Book xiii

What's in This Book xiii
Conventions Used in This Book xiv
 Special Fonts and Font Styles xiv
 Syntax Notation xiv
 Types of Notes xv
Development Environment xv
For More Information xv

Chapter 1

Introduction 1-1

About the Macintosh Debugger 1-3
 Basic Debugging Operations 1-4
 Debugging Different Types of Code 1-4
 Debugging Emulated Code 1-4
Debugger Components 1-5
 680x0 Source-Level Debugging 1-6
 PowerPC Source-Level Debugging 1-7
 Low-Level Debugging 1-8
The Debugger Interface 1-9

Chapter 2

Getting Started 1

Building Your Application for Debugging 2-3
Installing the 680x0 Source-Level Debugger 2-5
 Installing 2-5
 Launching 2-6
Installing the PowerPC Source-Level Debugger 2-7
 Installing 2-7
 Launching 2-8
Installing the Low-Level Nub 2-8
 Installing 2-9
 Specifying a Port 2-9
 Launching the Low-Level Debugger 2-10
 Beginning to Debug 2-11
 Reconnecting to the Target Machine 2-11
Changing the Debugger General Preferences 2-12

Working With the Debugger	3-3
Using the Browser Window	3-4
Displaying Multiple Code Views	3-6
Finding Code Quickly	3-6
Manipulating Windows	3-7
The Log Window	3-8
Printing the Contents of a Window	3-8
Taking a Snapshot of the Active Window	3-8
Remember Window Position	3-8
Setting and Clearing Breakpoints	3-9
Simple Breakpoints	3-10
One-Shot Breakpoints	3-11
Counting Breakpoints	3-12
Conditional Breakpoints	3-13
Performance Breakpoints	3-13
Displaying Breakpoints	3-14
Setting Breakpoints From Your Source Code	3-14
Disabling Calls to Debugger or DebugStr	3-15
Controlling Program Execution	3-15
Target ""	3-18
Untarget ""	3-18
Launch	3-18
Step to Branch	3-18
Step to Branch Taken	3-18
Looking at Memory	3-18
Searching Memory	3-20
Displaying PowerPC Instructions	3-21
Displaying 680x0 Instructions	3-22
The Stack	3-23
Registers	3-25
PowerPC General-Purpose Registers	3-25
680x0 General-Purpose Registers	3-26
PowerPC Floating-Point Registers	3-26
680x0 Floating-Point Registers	3-27
Displaying the Value of Variables	3-28
Names, Addresses, or Expressions	3-28
The Value of "this"	3-29
Global Variables	3-29
Handling Fragments	3-30
Show Fragment Info	3-31
Symbolic Mapping Preferences	3-32

Chapter 4

The Adaptive Sampling Profiler 4-1

About the Adaptive Sampling Profiler	4-3
Using the Adaptive Sampling Profiler	4-5
Enabling Performance Measurement	4-6
Starting a Profiling Session	4-6
Specifying a Sampling Rate	4-7
Collecting Performance Data	4-8
Measuring Selected Routines	4-8
Generating a Performance Report	4-9
How the ASP Presents Collected Data	4-9
The Summary View	4-11
The Statistics View	4-11
Editing the Performance Report	4-13
Saving and Printing a Performance Document	4-14
Evaluating Performance Data	4-15
How the Adaptive Sampling Profiler Gathers Data	4-15
Allocating Nodes	4-16
Splitting Buckets	4-17
Freeing Nodes	4-19
Registration Errors	4-19
ASP Overhead	4-19

Chapter 5

Low-Level Debugging 5-1

Memory	5-3
Data Storage	5-3
Data Alignment	5-4
Locating a Fragment's Code and Data in Memory	5-4
Obtaining Heap Information	5-7
Registers	5-7
The Stack	5-9
680x0 Stack Frames and Calling Conventions	5-12
PowerPC Stack Frames and Calling Conventions	5-12
System-Level Debugging	5-14
Debugging Memory-Based Fragments	5-14
Mixed-Mode Debugging	5-15

Chapter 6

Debugger Extensions 6-1

About Debugger Extensions	6-3
Writing a Debugger Extension	6-4
Sample Debugger Extension	6-6
Using Callback Routines	6-7
Building a Debugger Extension	6-8

Debugger Extension Reference	6-9
Constants	6-9
Data Structures	6-10
Callback Routines	6-11
Input Routines	6-11
Output Routines	6-13
Utility Routines	6-14

Appendix A	Expression Evaluation	A-1
------------	------------------------------	-----

Expression Grammar	A-3
Symbols	A-3
Constants	A-3
Register Names	A-3
Precedence of Operators	A-4
Expression Syntax	A-5
Error Messages	A-9

Appendix B	Debugger Menu Reference	B-1
------------	--------------------------------	-----

File Menu	B-3
Open...	B-3
Open ROM map	B-3
Map Symbolics to Code	B-3
Close	B-4
Save Log As...	B-4
Save Performance	B-4
Page Setup...	B-4
Print Window...	B-4
Quit	B-4
Edit Menu	B-5
Undo	B-5
Cut	B-5
Copy	B-5
Paste	B-6
Clear	B-6
Select All	B-6
Show Clipboard	B-6
General Preferences...	B-6
Target Preferences...	B-8
Symbolic Mapping Preferences...	B-9
Remember Window Position	B-11
Views Menu	B-11
Show Stack Crawl	B-11

Show Globals	B-13
Show Registers	B-14
Show FPU Registers	B-15
Show Memory	B-17
Show Instructions	B-18
Show 68K Instructions	B-19
Show Tasks	B-19
Show Fragment Info	B-20
Find Code For...	B-22
Find Code For ""	B-22
Source File For ""	B-22
Show Log Window	B-22
Control Menu	B-23
Run ""	B-24
Stop ""	B-24
Kill ""	B-24
Resume ""	B-24
Propagate Exception	B-24
Target ""	B-25
Untarget ""	B-25
Launch	B-25
Step	B-25
Step Into	B-25
Step Out	B-26
Step To Branch	B-26
Step To Branch Taken	B-26
Animate	B-26
Show Breakpoints List	B-26
Clear All Breakpoints	B-27
Show / Hide Control Palette	B-27
Evaluate Menu	B-27
Show Evaluation Window	B-28
Evaluate ""	B-28
Evaluate "this"	B-29
Displaying Values in Different Formats	B-29
Performance Menu	B-29
New Session	B-30
Show Source	B-31
Configure Utility	B-31
Configure Report	B-32
Enable / Disable Utility	B-33
Gather Report	B-33
The Summary View	B-34
The Statistics View	B-35
Extras Menu	B-36
Show PC	B-36
Search Memory	B-36

Enter MacsBug	B-37
Stop For DebugStrs	B-37
Execute Debugger Extension...	B-37
Snapshot Active Window	B-38
Windows Menu	B-38

Appendix C	Debugger Shortcuts	C-1
------------	---------------------------	-----

General	C-3
Browser Window	C-3
Breakpoints Window	C-3
Stack Window	C-4

Appendix D	Creating Custom Unmangle Schemes	D-1
------------	---	-----

Unmangle Schemes for 680x0 Code	D-3
Unmangle Schemes for PowerPC Code	D-5

Appendix E	Targeting Code Resources by Resource Type	E-1
------------	--	-----

About ResourceTracker	E-3
Using ResourceTracker	E-3

Glossary	GL-1
-----------------	------

Index	IN-1
--------------	------

Figures, Tables, and Listings

Chapter 1

Introduction 1-1

Figure 1-1	68K Mac Debugger components, two-machine source-level debugging	1-6
Figure 1-2	Power Mac Debugger components, two-machine source-level debugging	1-7
Figure 1-3	Power Mac Debugger components, low-level debugging	1-8
Figure 1-4	The browser's three panes	1-9
Figure 1-5	The control palette	1-10

Chapter 2

Getting Started 1

Figure 2-1	Building your application for debugging	2-4
Figure 2-2	The Debugger Nub Controls dialog boxes	2-10
Figure 2-3	Selecting your general preferences	2-12
Table 2-1	Files for debugging 680x0 source code	2-5
Table 2-2	Files for debugging PowerPC source code	2-7
Table 2-3	Files for low-level debugging (Power Macintosh target required)	2-9
Table 2-4	General Preferences options	2-13

Chapter 3

Using the Debugger 3-1

Figure 3-1	Parts of the browser window	3-4
Figure 3-2	A sample browser window	3-5
Figure 3-3	An alternative code view	3-6
Figure 3-4	The Find Code For dialog box	3-7
Figure 3-5	Resize icons	3-7
Figure 3-6	The log window	3-8
Figure 3-7	Setting a simple breakpoint	3-11
Figure 3-8	Setting a one-shot breakpoint	3-12
Figure 3-9	Selecting a counting breakpoint	3-12
Figure 3-10	Selecting a conditional breakpoint	3-13
Figure 3-11	Setting a performance breakpoint	3-13
Figure 3-12	Displaying breakpoints	3-14
Figure 3-13	The control palette	3-15
Figure 3-14	The PC in subroutine marker	3-17
Figure 3-15	The memory display using 4-byte alignment	3-19
Figure 3-16	The memory display using 1-byte alignment	3-20
Figure 3-17	Search Memory window	3-21
Figure 3-18	The PowerPC instruction display window	3-22
Figure 3-19	The 680x0 instruction display window	3-23
Figure 3-20	The stack display	3-24
Figure 3-21	The PowerPC registers window	3-25

Figure 3-22	The 680x0 registers window	3-26
Figure 3-23	The PowerPC floating-point registers	3-27
Figure 3-24	The 680x0 floating-point registers	3-27
Figure 3-25	The Evaluate dialog box	3-28
Figure 3-26	Displaying global variables	3-30
Figure 3-27	Fragment Info window	3-31
Figure 3-28	Show Exports window	3-32
Figure 3-29	Symbolic File to Fragment Name Mapping dialog box	3-32
Table 3-1	Debugger breakpoint icons	3-9
Table 3-2	Program control commands	3-16

Chapter 4

The Adaptive Sampling Profiler 4-1

Figure 4-1	Flat-time measurement	4-4
Figure 4-2	The Performance menu	4-5
Figure 4-3	A blank performance document window	4-6
Figure 4-4	The utility configuration dialog box	4-7
Figure 4-5	Selecting a performance breakpoint	4-8
Figure 4-6	A performance document window after a report has been gathered	4-10
Figure 4-7	Displaying data for a fragment	4-12
Figure 4-8	Data sorted by beginning address	4-13
Figure 4-9	The report configuration dialog box	4-14
Figure 4-10	Recording bucket hits in a node	4-17
Figure 4-11	Splitting a bucket	4-18

Chapter 5

Low-Level Debugging 5-1

Figure 5-1	Fragment Info window	5-6
Figure 5-2	Show Exports window	5-6
Figure 5-3	The stack display	5-10
Figure 5-4	Pascal and C stack-based calling conventions	5-12
Figure 5-5	The PowerPC stack frame	5-13
Table 5-1	The general-purpose registers	5-7
Table 5-2	The floating-point registers	5-8
Table 5-3	The Condition Register	5-8
Table 5-4	The special-purpose registers	5-9
Table 5-5	Stack frame information	5-11

Chapter 6

Debugger Extensions 6-1

Figure 6-1	Listing available debugger extensions	6-4
Table 6-1	<code>request</code> field values	6-5
Table 6-2	Header and library files for debugger extensions	6-8
Listing 6-1	Skeleton code for a debugger extension	6-5

Listing 6-2	Sample source code for debugger extension	6-6
Listing 6-3	Debugger extension makefile	6-8
Listing 6-4	Sample resource description file to create an 'ndcd' resource	6-9

Appendix A

Expression Evaluation A-1

Table A-1	PowerPC register names	A-4
Table A-2	680x0 register names	A-4
Table A-3	Precedence of operators	A-5

Appendix B

Debugger Menu Reference B-1

Figure B-1	The File menu	B-3
Figure B-2	The Edit menu	B-5
Figure B-3	The General Preferences dialog box	B-6
Figure B-4	Target preferences for Power Mac Debugger	B-9
Figure B-5	Target preferences for 68K Mac Debugger	B-9
Figure B-6	The Symbolic File to Fragment Name Mapping dialog box	B-10
Figure B-7	The Views menu	B-11
Figure B-8	The stack display	B-12
Figure B-9	The global variables display window	B-13
Figure B-10	The PowerPC registers display window	B-14
Figure B-11	The 680x0 registers window	B-15
Figure B-12	Displaying the PowerPC floating-point registers	B-16
Figure B-13	Displaying the 680x0 floating-point registers	B-16
Figure B-14	The memory display window	B-17
Figure B-15	The PowerPC instructions display window	B-18
Figure B-16	The 680x0 instructions display window	B-19
Figure B-17	The Task Browser window	B-20
Figure B-18	The Fragment Info window	B-21
Figure B-19	The Show Exports window	B-21
Figure B-20	The Find Code For dialog box	B-22
Figure B-21	The log window	B-23
Figure B-22	The Control menu	B-23
Figure B-23	The breakpoints window	B-26
Figure B-24	The Evaluate menu	B-28
Figure B-25	The Evaluate dialog box	B-28
Figure B-26	The Performance menu	B-30
Figure B-27	A blank performance document window	B-30
Figure B-28	The utility configuration dialog box	B-31
Figure B-29	The report configuration dialog box	B-32
Figure B-30	A performance document window after a report has been gathered	B-34
Figure B-31	The Extras menu	B-36
Figure B-32	The Search Memory window	B-37
Figure B-33	Listing available debugger extensions	B-38
Table B-1	General Preferences options	B-7
Table B-2	Control menu items and floating control palette icons	B-24

About This Book

This book, the *Macintosh Debugger Reference*, describes how you can use the Macintosh Debugger to debug your programs and, for PowerPC debugging, measure their performance.

This book is intended for developers creating applications, libraries, or other programs for Power Macintosh and 680x0-based Macintosh computers. It's also intended for developers converting existing 680x0 programs to run on Power Macintosh computers.

What's in This Book

You can think of the chapters and appendixes that make up this book as being divided into three parts. The first part, which includes Chapters 1 through 4, explains how to use the debugger to debug your program and measure its performance. You must read this part to use the debugger. The second part, which includes Chapters 5 and 6, addresses special topics related to low-level debugging and creating your own debugging extensions. The third part, the appendixes, consists of reference material. The contents of each chapter and appendix are summarized below:

- Chapter 1, "Introduction," describes the basic capabilities and general operation of the Macintosh Debugger.
- Chapter 2, "Getting Started," explains what you must do before you can use the debugger, including building your application for debugging and installing the debugger.
- Chapter 3, "Using the Debugger," explains how you use the debugger, focusing on source-level debugging.
- Chapter 4, "The Adaptive Sampling Profiler," explains how you use the Adaptive Sampling Profiler to measure how often sections of your code execute. (The Adaptive Sampling Profiler is not available on 68K Mac Debugger.)
- Chapter 5, "Low-Level Debugging," addresses special topics in interpreting the debugger's low-level displays.
- Chapter 6, "Debugger Extensions," describes how you can customize and extend the debugger by creating debugger extensions. (Debugger extensions are not available on 68K Mac Debugger.)
- Appendix A, "Expression Evaluation," describes the grammar of C and C++ expressions that can be evaluated using the debugger's expression evaluator.

- Appendix B, “Debugger Menu Reference,” is a complete reference guide to the debugger’s menu selections.
- Appendix C, “Debugger Shortcuts,” lists some shortcuts you can use with the Macintosh Debugger.
- Appendix D, “Creating Custom Unmangle Schemes,” outlines the steps necessary to create your own unmangle stand-alone code resource for use by the Macintosh Debugger.
- Appendix E, “Targeting Code Resources by Resource Type,” describes ResourceTracker, a small application that allows the user to target code resources that have a resource type other than the standard 'CODE ', such as 'INIT' or 'CDEV' or any resource type of your choice.

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

<i>Courier</i>	Throughout this book, words that you must type exactly as shown are in Courier. Courier 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:

<i>literal</i>	Courier text indicates a word that must appear exactly as shown. Special symbols (<i>∂</i> , <i>•</i> , <i>§</i> , <i>&</i> , and so on) must also be entered exactly as shown.
<i>italics</i>	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. ▲

Development Environment

All code listings in this book are shown in 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 *Muslin*. These are not actual products of Apple Computer, Inc.

For More Information

APDA 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 platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

P R E F A C E

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA

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
-----	--------------

AppleLink	APDA
-----------	------

America Online	APDAorder
----------------	-----------

CompuServe	76666,2405
------------	------------

Internet	APDA@applelink.apple.com
----------	--------------------------

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information of registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support

Apple Computer, Inc.

20525 Mariani Avenue, M/S 303-2T

Cupertino, CA 95014-6299

Introduction

Contents

About the Macintosh Debugger	1-3
Basic Debugging Operations	1-4
Debugging Different Types of Code	1-4
Debugging Emulated Code	1-4
Debugger Components	1-5
680x0 Source-Level Debugging	1-6
PowerPC Source-Level Debugging	1-7
Low-Level Debugging	1-8
The Debugger Interface	1-9

Introduction

This chapter provides an introduction to the **Macintosh Debugger**, a tool (consisting of two debuggers) to debug applications written for PowerPC-based and 680x0-based Macintosh computers at the assembler and source level. You can also use the debugger to trace your application's execution or, for PowerPC™ debugging only, to gather performance data. This chapter describes the basic capabilities and general operation of the debuggers.

After you've gotten an overview of the debugger—its capabilities, components, and program files—the last section of this chapter introduces you to the debugger interface, which lets you monitor and control the execution of your program. See Chapter 2, "Getting Started," for installation instructions.

About the Macintosh Debugger

The Macintosh Debugger performs high-level and low-level debugging. The terms *high level* and *low level* refer to the level at which you're connected to the nub (described in the next paragraph) when debugging, not to the type of code being debugged. Within this framework, you can perform assembler-level and source-level debugging.

Except for low-level debugging, you can use the Macintosh Debugger in a one-machine or two-machine environment. With two machines, the **host** side of the debugger is your user interface; you can put it on either of the two machines you're using. The **target**, or **nub**, side of the debugger runs on the other machine, along with the application you're debugging. This nub occupies very little memory and essentially just takes a snapshot of the state of the target machine and communicates this information to the host. The host, on the other hand, presents the information transmitted by the nub, using a variety of menus, windows, and dialog boxes that are tailored to make this information easily accessible and comprehensible. For one-machine debugging, the debugger uses the same files and has the same interface, but runs on one machine.

Power Mac Debugger provides source-level debugging of PowerPC code, and **68K Mac Debugger** provides source-level debugging of 680x0 code running on either a 680x0-based or a Power Macintosh computer. In a two-machine setup having at least one Power Macintosh computer, Power Mac Debugger also provides low-level debugging—of PowerPC source code and assembly-language code. Note that in low-level debugging, which is always on two machines, the target is frozen when it's communicating with the host.

Note

System software should be System 7.1 or later; System 7.5 is recommended. ♦

Basic Debugging Operations

The debugger provides basic debugging operations such as setting breakpoints, controlling program execution, and displaying the contents of variables. At the machine level, the debugger can also disassemble machine code, display and modify memory, and display and modify registers. You can use the debugger to

- display source or assembly code for a selected function
- read and write general-purpose and floating-point registers
- display the stack
- read from and write to memory
- disassemble PowerPC and 680x0 code
- display and edit local and global variables
- evaluate C and C++ expressions
- set debugging breakpoints or (for Power Mac Debugger only) performance breakpoints at the source or machine code level
- step through source or assembly code
- open multiple symbolic information files (shared library support)
- measure program performance (Power Mac Debugger only)

In addition, you can customize or extend the capabilities of Power Mac Debugger by writing debugger extensions for low-level debugging; these extensions are similar to the MacsBug 'dcmd' resources available with the MacsBug debugger.

Debugging Different Types of Code

You can use Macintosh Debugger to debug and measure the performance of applications, shared libraries, stand-alone code, and system software. To debug at the source code level, you need to have symbolic information for the code you want to debug loaded on the host machine. (Chapter 2 provides additional information about this process.) You do not need symbolic information to do machine-level debugging; however, keep in mind that in this case the host has limited symbolic information based on embedded strings if these are included in the code being debugged.

Debugging Emulated Code

If your PowerPC code calls an emulated Toolbox or Operating System routine, you can view information for that routine using Macintosh Debugger's stack window, which displays stack crawl information for both native and 680x0 stack frames. Power Mac Debugger can also disassemble 680x0 code and display memory. However, it cannot step or set breakpoints in 680x0 code.

Debugger Components

The subsections that follow describe the debugger components and program files you need to run the debugger; these required components are grouped in three sections:

- 680x0 source-level debugging components
- PowerPC source-level debugging components
- low-level debugging components (Power Macintosh target)

Each section gives you a simple view of the two-machine setups (in which you can see how the computers connect and where your files go) and then describes the necessary files. In the figures shown there, the phrase *680x0* refers to any Macintosh II (or higher) computer. However, for the most efficient processing, a Macintosh Quadra computer is recommended. The phrase *680x0 or Power Macintosh* refers to any Macintosh II (or higher) computer, including Power Macintosh. For the most efficient processing, a Macintosh Quadra or Power Macintosh computer is recommended.

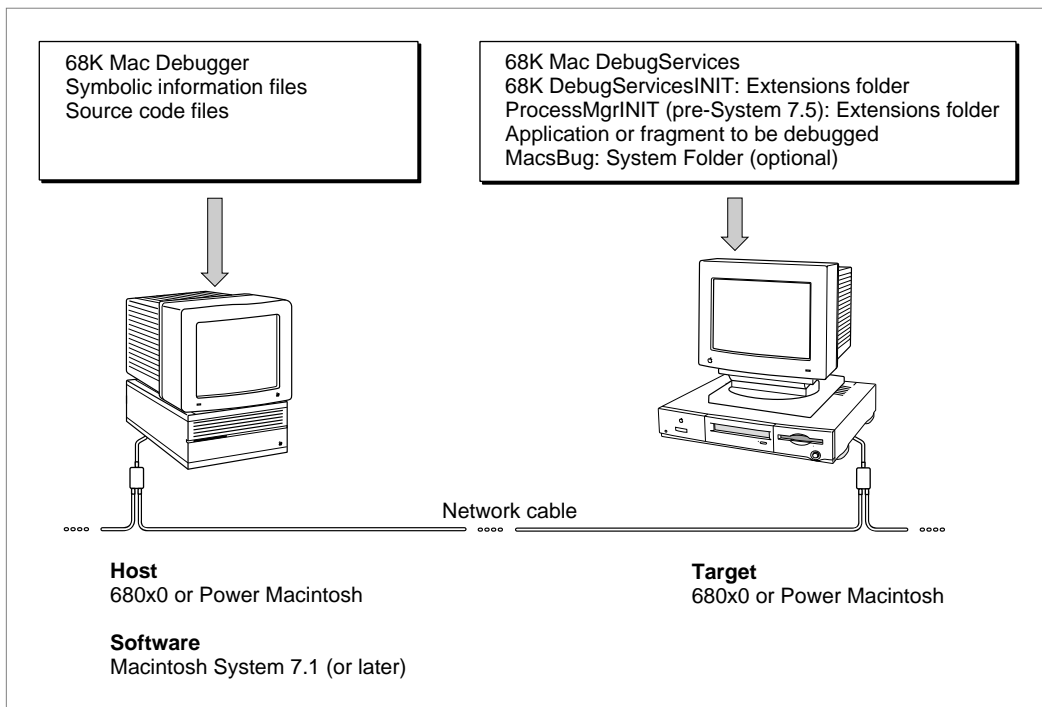
Besides the required debugger files listed in the subsections that follow, there are optional files, listed here to avoid repeating them within each section:

- **an optional preferences file.** If you have written debugger extensions, you'll need to create a preferences file for them. For Power Mac Debugger, name this file *Power Mac DebugServices Prefs*, and for 68K Mac Debugger, name it *68K DebugServices Prefs*. If you have written a custom C++ unmangle scheme, add it to the *Macintosh Debugger Preferences* folder.
- **MacsBug.** You can use Macintosh Debugger without installing MacsBug, but having it installed lets you gather information following a crash in emulated code, display information about system data structures, and set breakpoints on emulated system calls. For information about MacsBug, see the *MacsBug Reference and Debugging Guide*.
- **an extension for systems earlier than System 7.5.** If your system software is earlier than 7.5, you must install an extension. For 680x0 machines, install the extension *ProcessMgrINIT*; for Power Macintosh machines, install the extension *PPCTraceEnabler*.

680x0 Source-Level Debugging

You can debug 680x0 source code using either one or two machines. Remember that the name 68K Mac Debugger refers to the type of code being debugged, not to the type of machine. Therefore you may use a 680x0-based or Power Macintosh computer or any combination. For two machines (see Figure 1-1), some debugger files are put on the host and some on the target. For one machine, the same files are used but the host-target distinction is not important.

Figure 1-1 68K Mac Debugger components, two-machine source-level debugging



You need the following files to do 680x0 source-level debugging:

- **68K Mac Debugger.** This application is the user interface.
- **68K Mac DebugServices.** This background application is the debugger nub; it collects the information from your application.
- **68K DebugServicesINIT.** This extension contains nub code.
- **68K Mac Debugger Prefs.** This file is created automatically when you launch the debugger. It works with the debugger and includes information about default debugger settings. You can change these settings by choosing General Preferences from the Edit menu.
- **Symbolic information file.** You create a symbolic information file when you build your application for debugging.

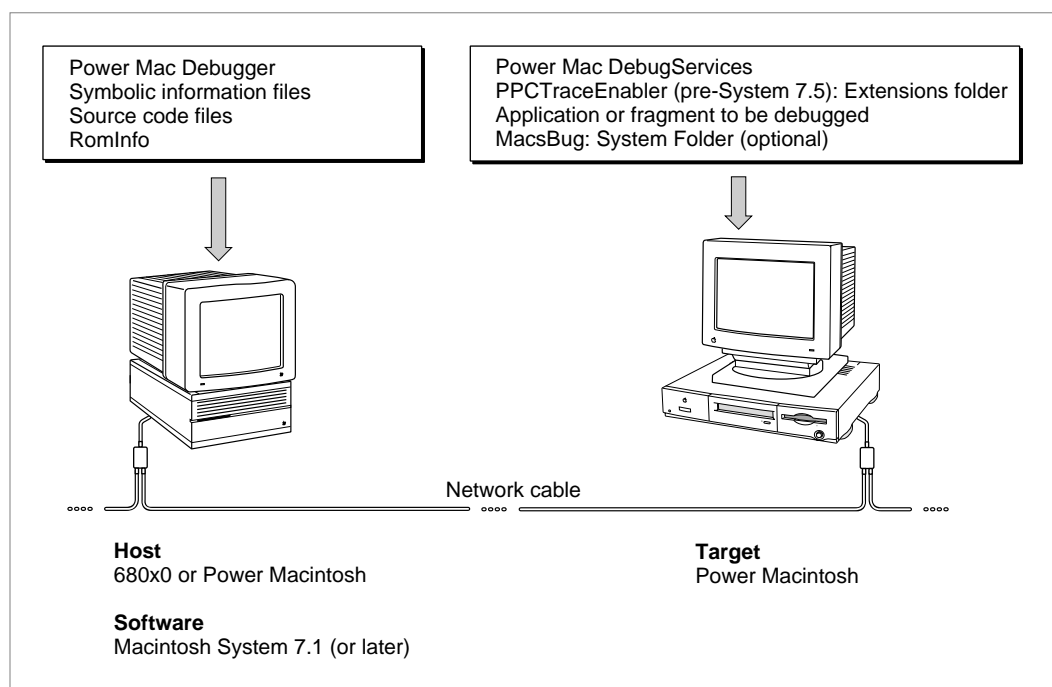
Introduction

- **Source code files.** These are your program's source code files, which must be accessible to do source-level debugging.

PowerPC Source-Level Debugging

You can debug PowerPC source code using either one or two machines. For two machines (see Figure 1-2), some files are put on the host and some on the target. For one machine, the host-target distinction is not important. If you are using two machines, the host can be either a 680x0-based Macintosh or a Power Macintosh computer.

Figure 1-2 Power Mac Debugger components, two-machine source-level debugging



You need the following files to do PowerPC source-level debugging:

- **Power Mac Debugger.** This application is the user interface.
- **Power Mac DebugServices.** This background application is the debugger nub; it collects the information from your application.
- **Power Mac Debugger Prefs.** This file, created automatically when you launch the debugger, works with the debugger and gives you information about default debugger settings. You can change these settings by choosing General Preferences from the Edit menu.
- **RomInfo file.** The RomInfo file provides information to the debugger about PowerPC code stored in ROM.

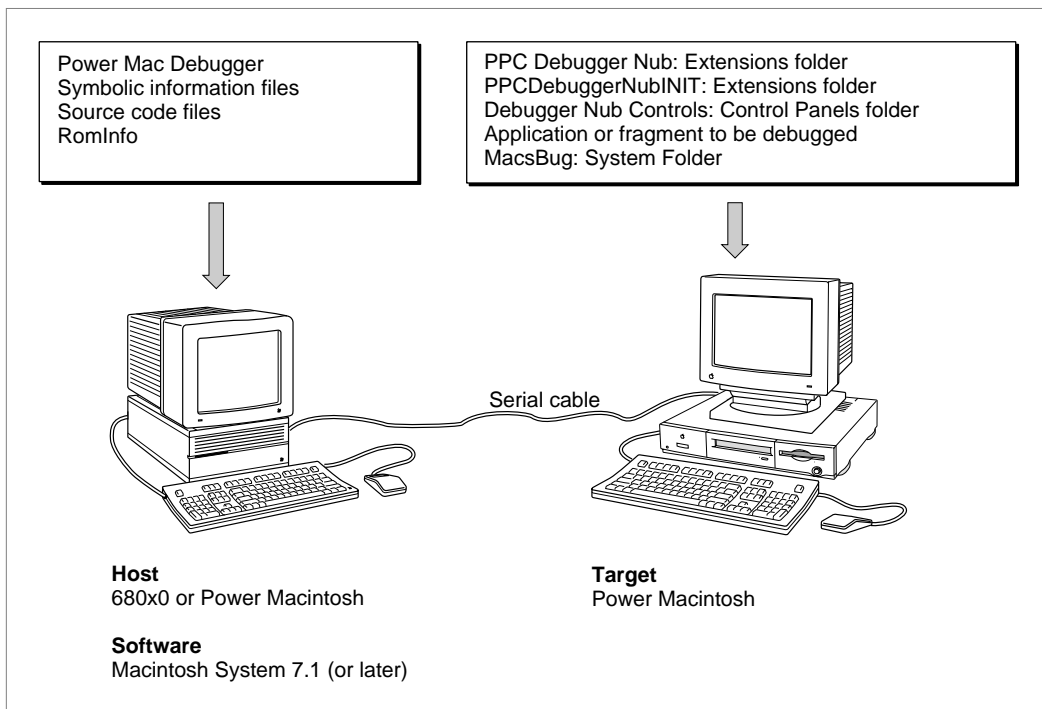
Introduction

- **Symbolic information file.** You create a symbolic information file when you build your application for debugging.
- **Source code files.** These are your program's source code files, which must be accessible to do source-level debugging.

Low-Level Debugging

Low-level debugging is available for the Power Macintosh using two machines (see Figure 1-3). Here the nub is a low-level debugger that disables interrupts on the target machine, preventing continuous communication between your computer and a file server. To avoid delays while your computer waits for connections with a server to be timed out, you should dismount all server volumes.

Figure 1-3 Power Mac Debugger components, low-level debugging



The necessary files, listed here, are described in terms of the host computer and target (nub) computer.

- **Power Mac Debugger.** This application is the user interface.
- **PPC Debugger Nub.** This extension contains nub code.
- **PPCDebuggerNubINIT.** This extension also contains nub code.

Introduction

- **Power Mac Debugger Prefs.** This file, created automatically on the host when you launch the debugger, works with the debugger and includes information about default debugger settings. You can change these settings by choosing General Preferences from the Edit menu.
- **Debugger Nub Controls.** This file is a control panel on the target computer. When you open Debugger Nub Controls, a primary configuration window is displayed that allows you to specify which port is to be used and whether the nub is active.
- **RomInfo file.** The RomInfo file, which goes in the same folder as Power Mac Debugger, provides information to the debugger about PowerPC code stored in ROM.

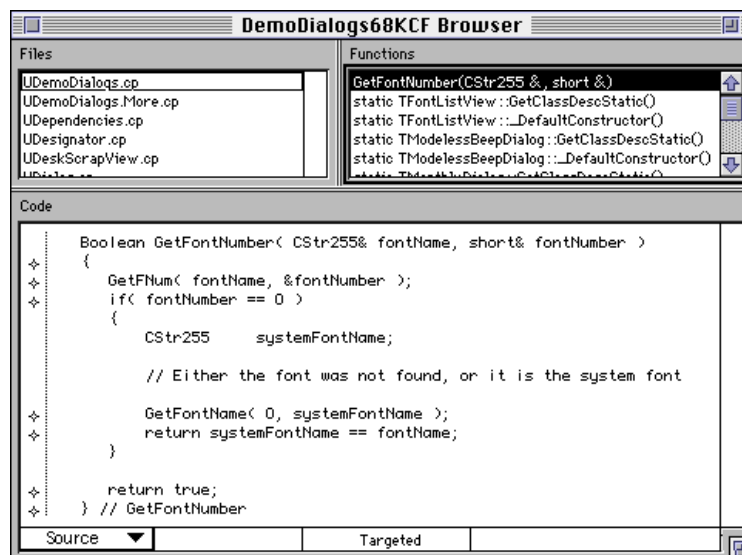
The Debugger Interface

The Macintosh Debugger interface consists of menus, windows, dialog boxes, and a floating control palette, all of which allow you to monitor and control the execution of your application. Use the debugger to run your application under controlled conditions and to determine and change its state at any given point in time.

The debugger's two main control posts are the **browser window** and the **control palette**. You use the browser window to navigate through your source code and to set and clear breakpoints. You use the control palette (or equivalent menu commands) to control the execution of your program.

Figure 1-4 shows a browser window for the sample application DemoDialogs. This three-pane window allows you to select a source file (upper-left pane) and a function in that source file (upper-right pane). You view the source code for the selected function in the lower pane, also called the code view.

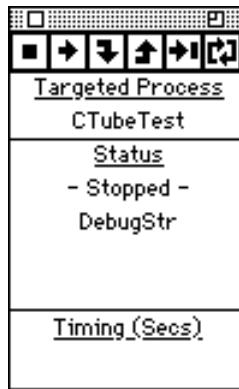
Figure 1-4 The browser's three panes



Introduction

Figure 1-5 shows the control palette for the sample application CTubeTest. To stop or run the target program, you click the program-control icons at the top of the palette. The control palette also displays information about the target process and its current status, and if the process is stopped, it specifies the cause of the break and the time elapsed since the last break.

Figure 1-5 The control palette



See Chapter 3, “Using the Debugger,” for more information on the browser window and the control palette.

Getting Started

Contents

Building Your Application for Debugging	2-3
Installing the 680x0 Source-Level Debugger	2-5
Installing	2-5
Launching	2-6
Installing the PowerPC Source-Level Debugger	2-7
Installing	2-7
Launching	2-8
Installing the Low-Level Nub	2-8
Installing	2-9
Specifying a Port	2-9
Launching the Low-Level Debugger	2-10
Beginning to Debug	2-11
Reconnecting to the Target Machine	2-11
Changing the Debugger General Preferences	2-12

Getting Started

This chapter explains what you must do before you can use the Macintosh Debugger to debug and profile your application. In general, this includes building your application for source-level debugging and then installing and launching the debugger.

The installation and launch discussion is organized, like Chapter 1, in three logical groupings:

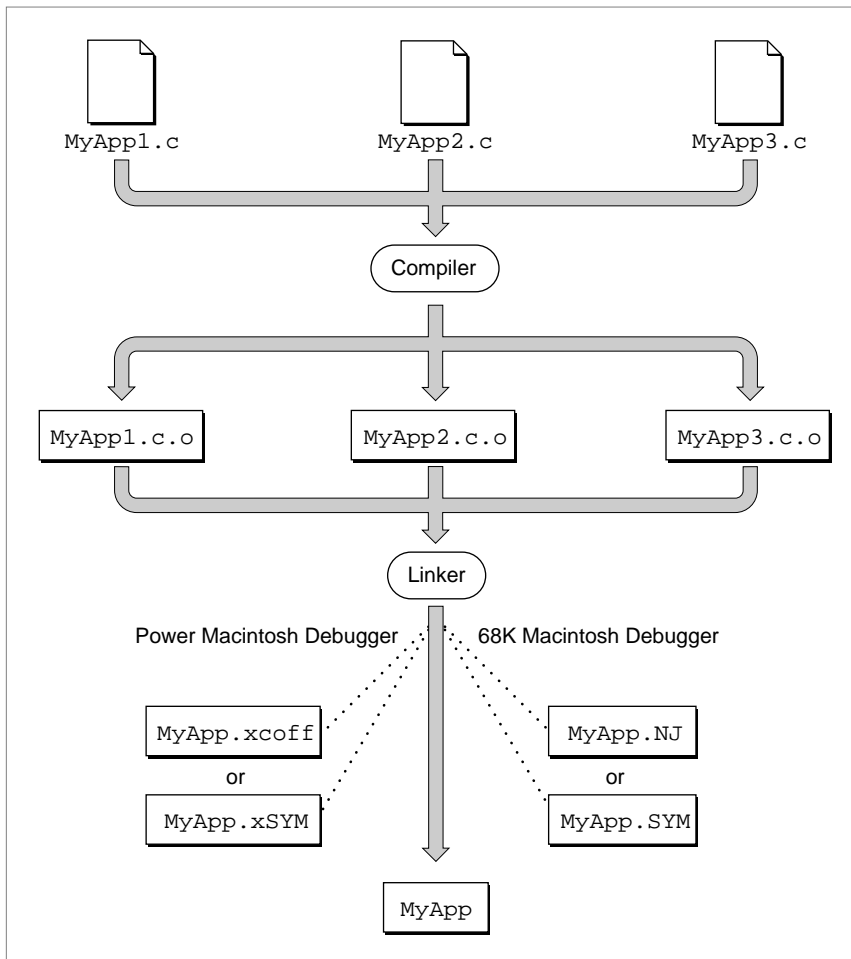
- 68K Mac Debugger and associated files, to debug 680x0 code
- Power Mac Debugger and associated files, to debug PowerPC code
- Power Mac Debugger and associated files, to do low-level debugging of PowerPC code

After you have an understanding of the material presented in this chapter, you can use the debugger either to debug your application, as described in Chapter 3, “Using the Debugger,” or, for Power Mac Debugger, to measure your application’s performance, as described in Chapter 4, “The Adaptive Sampling Profiler.”

Building Your Application for Debugging

Building your application for debugging consists in gathering symbolic information during the build process and building a **symbolic information file** containing this information.

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

Figure 2-1 Building your application for debugging

Power Mac Debugger works with `.xcoff` and `.xSYM` files; 68K Mac Debugger works with `.NJ` and `.SYM` files.

When you compile your source code files, you need to make sure that the compiler produces correct symbolic information. You do this by specifying the option that produces symbol information. The debugger uses a symbolic map of a compiled and linked application to determine which assembled instructions correspond to the statements in your application's source code. This mapping is what makes debugging at the source code level possible.

Note

Be sure that the `.xcoff` files are linked with the option `-SYM big`. Otherwise, variable evaluation and linker speed are considerably slower. You also need to turn off all optimization when compiling your modules. If you generate optimized code, you cannot do source-level debugging. ♦

Installing the 680x0 Source-Level Debugger

Here are the steps for installing and launching 68K Mac Debugger on one or two machines. Apple recommends a Macintosh Quadra or higher for your 680x0-based machine, running Macintosh system software 7.1 or later. System 7.5 is preferred.

If you use ILink to prepare files for 68K Mac Debugger, be aware that the state (.NJ) file format uses full pathnames to the object files to get symbolic information. If an object file is moved, deleted, or changed in any way, you'll get an error from 68K Mac Debugger when reading the state file. Also, if you move the state file to another machine, even if that machine has the same sources as the original machine, the debugger will fail when reading the state file. This means that you must do the build on the machine on which you plan to run 68K Mac Debugger. All of the files listed in this section are described in Chapter 1.

Note

If you use two machines, be sure to enable program linking on the target machine through the control panels Sharing Setup and Users & Groups. ♦

Installing

Install the files listed in Table 2-1.

Table 2-1 Files for debugging 680x0 source code

Filename	Locate file	If two machines
68K Mac Debugger	Anywhere	On host
68K Mac DebugServices*	In System Folder's Startup folder	On target
68K DebugServicesINIT†	In Extensions folder	On target

* If installed in the System Folder, this application launches automatically on startup; if it is installed elsewhere, launch it by double-clicking it if you are on two machines. (If you are on one machine, it is launched when you launch the debugger.)

† This file is incompatible with other debugger INITs. Remove INITs from other debuggers that may be on your system before attempting to use 68K Mac Debugger.

Install the following additional files if necessary or desired. If you are using two machines, these files go on the target computer.

- ProcessMgrINIT, if your system software is earlier than 7.5. Put it in the Extensions folder. To do this, drag it to the System Folder icon. Note that ProcessMgrINIT is not compatible with Drag Enabler.
- 68K DebugServices Prefs, if you have written C++ unmangle schemes. Put this file in the System Folder's Preferences folder.

Getting Started

- ☐ MacsBug, if you want to gather additional information (see page 1-5). Install MacsBug in the System Folder.

Reboot your (target) machine; 68K Mac DebugServices will start automatically if you've put it in the System Folder.

Note

The nub 68K Mac DebugServices, because it's a background application, does not appear as an item in the Application menu. To quit the nub when you've finished debugging, press the keys Shift-Control-Delete; the dialog box Quitting DebugServices appears. Click OK to quit. ♦

Launching

1. Launch 68K Mac Debugger. The target connection dialog box appears.

When you launch the (host) debugger for the first time, the 68K Mac Debugger Prefs file is created. This file includes information about default debugger settings.

2. Select the target connection type.

For one-machine debugging, choose the radio button Local; for two-machine debugging, choose the radio button AppleTalk. Click OK.

3. Select the target machine (if using two machines).

After you select the target connection type and click OK, a dialog box appears, from which you select the target machine. The list displays only computers currently running the 68K Mac DebugServices nub. Select the desired machine and click OK. A Program Linking dialog box now asks for your user name and password. Enter them and click OK. A standard file dialog box appears.

4. Select the application to use for the debugging session or select a .NJ or .SYM file.

If you select your application, 68K Mac Debugger loads the appropriate symbolic information file (.NJ if available, or .SYM), launches the application, and does a break on launch automatically. The application you are debugging must have a 'SIZE' resource, and the CanBackground bit within this resource must be set to TRUE.

If you select a .NJ or .SYM file instead of the application, target the application by using one of the following methods:

- ☐ Cause an exception in the application. To do this, hold down the Control key when launching the application. This causes a code load exception when the application is launched. (This is the same as selecting the application by choosing the Open command in the File menu or choosing the Launch command in the Control menu.)
- ☐ Include Debugger and DebugStr calls in your program. You can make the call the first executable statement of your program, or you can include it anywhere in your source code file that is convenient for you.
- ☐ You can also target a running process, as described in "Controlling Program Execution," in Chapter 3.

Installing the PowerPC Source-Level Debugger

Here are the steps for installing and launching Power Mac Debugger on one or two machines.

Note

If you use two machines, be sure to enable program linking on the target machine through the control panels Sharing Setup and Users & Groups. ♦

Installing

Install the files listed in Table 2-2.

Table 2-2 Files for debugging PowerPC source code

Filename	Locate file	If two machines
Power Mac Debugger	Anywhere	On host
RomInfo	In folder with Power Mac Debugger	On host
Power Mac DebugServices*	In System Folder's Startup folder	On target

* If installed in System Folder, this application launches automatically on startup; if installed elsewhere, you must launch it by double-clicking it.

Install the following additional files if necessary or desired. If you are using two machines, these additional files go on the target machine.

- PPCTraceEnabler, if your system software is earlier than 7.5. Put it in the Extensions folder. To do this, drag it to the System Folder icon.
- Power Mac DebugServices Prefs, if you have written debugger extensions or C++ unmangle schemes. Put this file in the System Folder's Preferences folder.
- MacsBug, if you want to gather additional information (see page 1-5). Install MacsBug in the System Folder.

Reboot your (target) machine; Power Mac DebugServices will start automatically if you've put it in the System Folder.

Note

The nub Power Mac DebugServices, because it's a background application, does not appear as an item in the Application menu. To quit the nub when you've finished debugging, press the keys Shift-Control-Delete; the dialog box Quitting DebugServices appears. Click OK to quit. ♦

Launching

1. Launch Power Mac Debugger. A target connection dialog box appears.

When you launch the debugger the first time, the Power Mac Debugger Prefs file is created. This file includes information about default debugger settings.

2. Select the target connection type.

For one-machine debugging, choose the radio button Local; for two-machine debugging, choose the radio button AppleTalk. Click OK.

3. Select the target machine (if using two machines).

After you select the target connection type and click OK, a dialog box appears, from which you select the target machine. The list displays only Power Macintosh computers currently running the Power Mac DebugServices nub. Select the desired machine and click OK. A Program Linking dialog box now asks for your user name and password. Enter them and click OK. A standard file dialog box appears.

4. Select the application to use for the debugging session or select a `.xcoff` or `.xSYM` file.

If you select your application, Power Mac Debugger loads the appropriate symbolic information file (`.xcoff` if available, or `.xSYM`), launches the application, and does a break on launch automatically. The application you are debugging must have a 'SIZE' resource, and the CanBackground bit within this resource must be set to TRUE.

If you select a `.xcoff` file or a `.xSYM` file instead of the application itself, target the application by using one of the following methods:

- ☐ Cause an exception in the application. To do this, hold down the Control key when launching the application. This causes a code load exception when the application is launched. (This is the same as selecting the application using the Open command in the File menu or choosing the Launch command in the Control menu.)
- ☐ Include `Debugger` and `DebugStr` calls in your program. You can make the call the first executable statement of your program, or you can include it anywhere in your source code file that is convenient for you.
- ☐ You can also target a running process, as described on page 3-18.

Installing the Low-Level Nub

In debugging at the machine level, the host portion (Power Mac Debugger) stays the same but the target portion is different. This section gives the steps for installing the low-level nub.

IMPORTANT

Use the System Peripheral 8 Cable (part number M0197LL/B) to connect the host and target because other cables might result in communications failures. Because of the high speed of this serial connection, you should avoid running the cable near monitors or power cables. ▲

Getting Started

Note that if you are debugging and measuring code that has been optimized for speed, the compiler may inline some functions. The resulting inlined functions cannot have timing information generated for them nor can they have breakpoints placed in them.

Installing

Install the files listed in Table 2-3.

Table 2-3 Files for low-level debugging (Power Macintosh target required)

Filename	Locate file	Machine
Power Mac Debugger	Anywhere	On host
RomInfo	In folder with Power Mac Debugger	On host
PPCDebuggerNubINIT	In Extensions folder	On target
PPC Debugger Nub	In Extensions folder	On target
Debugger Nub Controls	In Control Panels folder	On target

Install the following additional files if necessary or desired. These files go on the target machine.

- Debugger Nub Preferences, if you have written debugger extensions or C++ unmangle schemes. Put this file in the System Folder's Preferences folder.
- MacsBug, if you want to gather additional information (see page 1-5). Install MacsBug in the System Folder.

Reboot your target machine to activate PPC Debugger Nub, and then disconnect all server volumes on it.

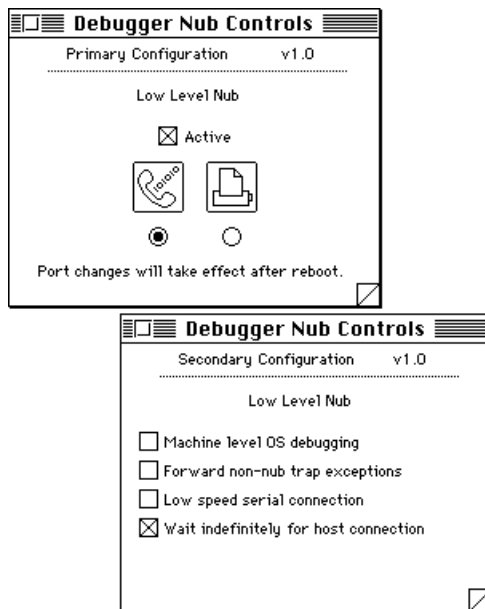
Note

For most efficient performance, do not have PPC Debugger Nub (the low-level nub) and Power Mac DebugServices (the high-level nub) active at the same time. ♦

Specifying a Port

To specify a port for your target machine, open the Debugger Nub Controls file (in the target's Control Panels folder). When you open this file, a primary configuration dialog box is displayed that allows you to specify which port you want used and that tells you whether the nub is active. If you click the page-turning control (in the lower-right corner of the dialog box), a secondary configuration window is displayed that allows you to make additional selection. Figure 2-2 shows both Debugger Nub Controls dialog boxes.

Getting Started

Figure 2-2 The Debugger Nub Controls dialog boxes

These are the selections you can make:

- **Machine level OS debugging.** This setting is reserved. If it is selected, it is not possible for the nub to make calls to the Code Fragment Manager or to execute a "break on launch." You will be able to do machine-level debugging, but source-level features will not function correctly.
- **Forward non-nub trap exceptions.** This item is used only for debugging the debugger.
- **Low-speed serial connection.** Check this box if you are using a host Macintosh 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.** If you do not check this box, the debugger times out after attempting to connect to the host for 5 seconds.

Launching the Low-Level Debugger

1. Launch Power Mac Debugger. A target connection dialog box appears.

When you launch the debugger, the Power Mac Debugger Prefs file is created. This file includes information about default debugger settings.

2. Select the target connection type.

Choose the radio button Remote, and then choose the radio button representing the port to which you connected the serial cable (either Modem Port or Printer Port). Click OK.

Getting Started

A standard file dialog box appears.

3. **From the standard file dialog box, select the symbolic information file (either `.xcoff` or `.xSYM`) for the application you want to debug.**

If you want to do low-level debugging without symbolic information, click Cancel.

Beginning to Debug

You can begin to do low-level debugging in two steps, as described next.

1. **Stop the application you want to debug on the target machine.**

To enable communication between the debugger nub and the host and to begin a debugging session, you must stop the application you want to debug on the target machine. This allows you to enter the debugger and specify the target process.

2. **Enter the debugger you have already launched, then specify the target process in one of the following ways:**

- ☐ Hold down the Control key when launching the application. This executes a break-on-launch function and allows you to enter the debugger as soon as your application is launched.
- ☐ Click the Stop icon (black square) from the control palette or choose Stop from the Control menu. (Make sure that the target is the current process when you choose Stop; for example, pull and hold down a menu in the target application while clicking Stop.)
- ☐ Include `Debugger` or `DebugStr` calls in your program. You can make the call the first executable statement of your program, or you can include it anywhere in your source code file that is convenient for you.

Once you have entered the debugger, you can use all the features of the debugger to look at the stack and registers, to display and change memory, to run your program under controlled conditions, and so on.

It's possible to debug several applications at one time. You enter the debugger for each application in the normal manner, as described above. Once you stop in the debugger in one application, however, the target machine is effectively frozen. The other application receives no processing time from the Process Manager. To the debugger, however, the other application appears to be running (because the debugger hasn't stopped it). Any windows specific to that application (for instance, its stack or registers) are not updated. Only the stopped process (the one whose name is displayed in the control palette) is current.

Reconnecting to the Target Machine

If the host loses its connection to the target machine when you are doing low-level debugging, how you reconnect depends on whether the target process is stopped:

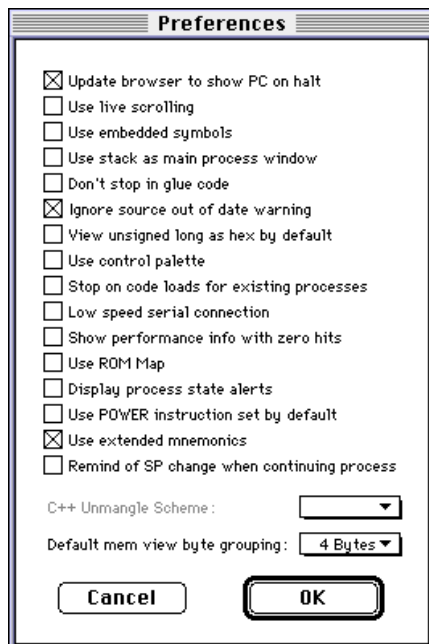
- If the target process is stopped, you need to relaunch the debugger on the host machine.
- If the target process is not stopped, you need to stop it to permit the host and nub to get back in sync and resume communication.

Changing the Debugger General Preferences

You may want to change some features of the debugger interface and of its memory display before you start debugging. To do this, choose General Preferences from the Edit menu and select the options you desire. Figure 2-3 shows the dialog box that appears, from which you can select your desired preferences. Note that not all general preferences are available for 68K Mac Debugger (see Table 2-4 for those that are not).

To define the features initially, the debugger uses values stored in the preferences file that was created automatically on the host machine when you first launched the debugger.

Figure 2-3 Selecting your general preferences



Getting Started

Table 2-4 describes the General Preferences options.

Table 2-4 General Preferences options

Option	Effect when selected
Update browser to show PC on halt	Updates the source code display in the browser so that the current program counter is displayed whenever the program is stopped.
Use live scrolling	Scrolls windows as the scroll box moves in the scroll bar. Otherwise, scrolling occurs when the scroll box stops moving.
Use embedded symbols	Displays function name symbols in the stack and instruction windows when no symbolic information file is found. 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 down the debugger.
Use stack as main process window	Uses the stack window rather than the registers window as the main process window. The debugger must have an active, unclosable window at all times. If you do not select this option, the debugger uses the registers window as this “main process” window by default. You must relaunch the debugger to have this option take effect.
Don’t stop in glue code	The debugger stops only in code that has symbolic information mapped to it. The debugger doesn’t stop in glue code.
Ignore source out of date warning	Eliminates “Source out of date” warnings. The debugger compares modification dates to determine if a symbolic information file is out of date with respect to the target application or the source files. For example, if you recompile the executable file without recreating the symbolic information file or if you delete comments from one of your source files, the debugger alerts you to the fact that the files’ modification dates no longer match. You can safely ignore this message if you know that whatever changes have been made will not affect the mapping between files.
View unsigned long as hex by default	Displays unsigned long values as hexadecimal values in the stack crawl, variable, and global variables windows. Leaving this option unselected displays unsigned long values as unsigned decimal values.
Use control palette	Causes the control palette to be displayed when the debugger is launched.
Stop on code loads for existing processes	Stops a targeted process when it loads any code, for example, when an application loads a code fragment.
Low speed serial connection	<i>Not available for 68K Mac Debugger.</i> Check this box if your host machine is slower than a Macintosh Quadra and you are experiencing communication problems with the target machine. If you check this box, you must also check the low-speed serial connection box in the secondary configuration window of the Debugger Nub Controls dialog box.
Show performance info with zero hits	<i>Not available for 68K Mac Debugger.</i> Instructs the Adaptive Sampling Profiler (ASP) to display buckets with zero hits in a performance report. Leave this option unselected to suppress the display of all empty buckets.

Table 2-4 General Preferences options (continued)

Option	Effect when selected
Use ROM Map	When you check this box, the debugger uses a ROM map. The first time the nub tries to connect to the host, the debugger requests the location of the ROM map through a standard file dialog box. The debugger then saves the location of the ROM map file and opens it automatically on subsequent launches of the host.
Display process state alerts	Check this box to display all informational alerts for a process. When you do not check the box, then some of the alerts are not displayed; for example, no alert appears when the debugger stops due to a break on launch.
Use POWER instruction set by default	<i>Not available for 68K Mac Debugger.</i> Displays POWER mnemonics in the instruction window. If you leave this box unchecked, the debugger displays PowerPC mnemonics.
Use extended mnemonics	<i>Not available for 68K Mac Debugger.</i> When you check this box, Power Mac Debugger disassembles PowerPC code using the extended mnemonic set. (If you selected the Use POWER instruction set by default option, then this preference does nothing.) If you do not check this box, then the debugger does not use extended mnemonics.
Remind of SP change when continuing process	<i>Not available for 68K Mac Debugger.</i> You can change the stack pointer in order to stack crawl from any location. If you then step or run the process, a crash is likely to occur. If this box is checked, the debugger reminds you of this and gives you the option to reset the stack pointer to the original value before stepping or running the process.
C++ Unmangle Scheme	Use this option to select a custom C++ unmangle scheme that you have previously added to the Macintosh Debugger Preferences folder. Refer to Appendix D, “Creating Custom Unmangle Schemes,” for more information on unmangle schemes.
Default mem view byte grouping	Sets the default memory display. Choose a value from the pop-up menu to set the default memory display at 4-byte grouping, 2-byte grouping, or 1-byte grouping.

Using the Debugger

Contents

Working With the Debugger	3-3
Using the Browser Window	3-4
Displaying Multiple Code Views	3-6
Finding Code Quickly	3-6
Manipulating Windows	3-7
The Log Window	3-8
Printing the Contents of a Window	3-8
Taking a Snapshot of the Active Window	3-8
Remember Window Position	3-8
Setting and Clearing Breakpoints	3-9
Simple Breakpoints	3-10
One-Shot Breakpoints	3-11
Counting Breakpoints	3-12
Conditional Breakpoints	3-13
Performance Breakpoints	3-13
Displaying Breakpoints	3-14
Setting Breakpoints From Your Source Code	3-14
Disabling Calls to Debugger or DebugStr	3-15
Controlling Program Execution	3-15
Target ""	3-18
Untarget ""	3-18
Launch	3-18
Step to Branch	3-18
Step to Branch Taken	3-18
Looking at Memory	3-18
Searching Memory	3-20
Displaying PowerPC Instructions	3-21
Displaying 680x0 Instructions	3-22
The Stack	3-23
Registers	3-25

PowerPC General-Purpose Registers	3-25
680x0 General-Purpose Registers	3-26
PowerPC Floating-Point Registers	3-26
680x0 Floating-Point Registers	3-27
Displaying the Value of Variables	3-28
Names, Addresses, or Expressions	3-28
The Value of “this”	3-29
Global Variables	3-29
Handling Fragments	3-30
Show Fragment Info	3-31
Symbolic Mapping Preferences	3-32

Using the Debugger

This chapter explains how you use the debugger, focusing on source-level debugging. Because no two developers are likely to follow the same sequence of steps in debugging an application, this chapter describes the debugger's interface using topic-based sections rather than a tutorial approach. If you cannot find help for a specific issue, you can also check Appendix B, "Debugger Menu Reference." Chapter 5, "Low-Level Debugging," offers additional information on interpreting data presented in the debugger's memory, registers, and stack windows.

This chapter explains how you use the debugger to

- display source and assembly views of selected routines
- set breakpoints and control program execution
- display the value of PowerPC and 680x0 registers
- disassemble PowerPC and 680x0 code
- display stack frames
- display the value of variables

The debugger also includes a performance measurement utility called the Adaptive Sampling Profiler. This utility is described in Chapter 4, "The Adaptive Sampling Profiler."

Working With the Debugger

When you first launch the debugger and stop your application, the debugger automatically displays the following windows:

- A browser window. You use this window to display source or assembly views of a function in any of your application's source files.
- A registers window. You use this window to examine the values stored in the PowerPC or 680x0 registers.

You can have the debugger open the stack window instead of the registers window at startup by checking the appropriate box in the General Preferences dialog box, available from the Edit menu. You must relaunch the debugger to have this option take effect.

- A control palette. You can use this window to control program execution.
- A log window. The debugger opens this window if you have entered the debugger using a call to `DebugStr`. The string you specified as a parameter to `DebugStr` is displayed in the log window.

Note that the browser window and the registers (or stack) window are associated with the symbolic information file and the target process, respectively. If you close the browser window, all the symbolic information for that symbolic information file is removed from any other window associated with it. You cannot close the registers (or stack) window once you have selected it to be the window to be opened at launch time.

Using the Debugger

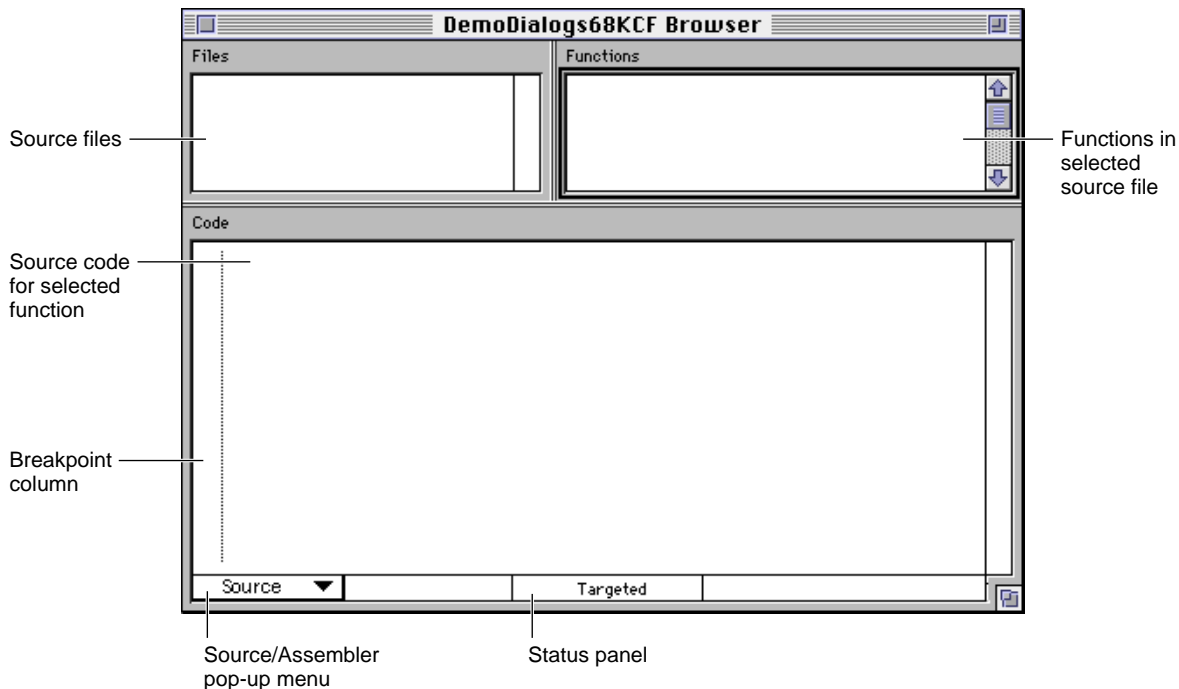
IMPORTANT

The debugger uses a variety of windows to allow you to navigate through source code, display different views of memory, and evaluate variables. Because all open windows are updated, having more windows open than you need affects the performance of the debugger. To improve performance, close unnecessary windows. ▲

Using the Browser Window

One of the windows displayed at startup (when you open a symbolic information file) is a three-pane window called a **browser window**. Figure 3-1 shows a labeled schematic representation of the browser window.

Figure 3-1 Parts of the browser window



The title of the window is the name of the application followed by the word “Browser.” The upper-left pane displays a list of source files for the symbolic information file. To select a source file, click its name. Once you have selected a file, the names of the functions referenced in the file are displayed in the upper-right pane. The lists in both upper panes are in alphabetical order.

To select a function, click the function name. Once you have selected a function, the source code for the function is displayed in the lower pane. This pane is called the **code view**. If no source file is available for the selected function (because, for example, the function is imported from an application or shared library that is not compiled with the

Using the Debugger

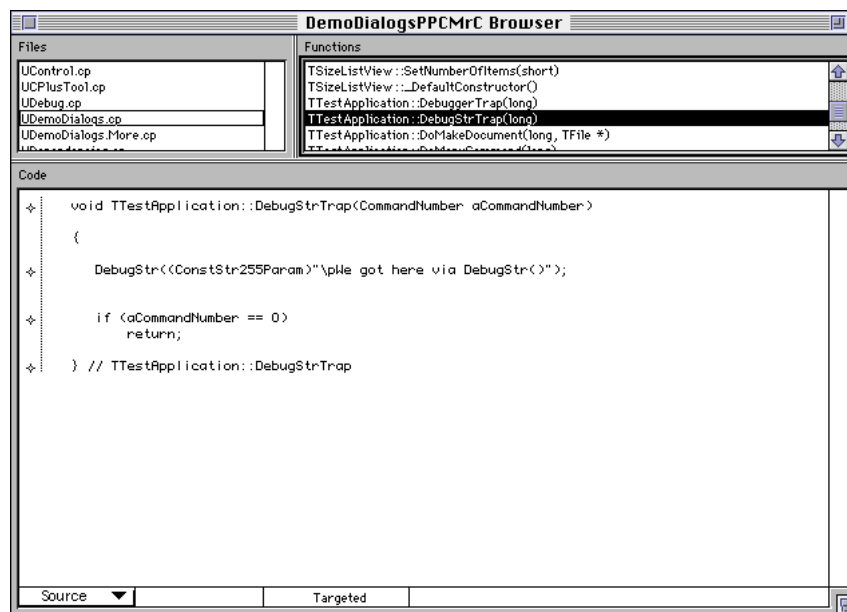
–sym on option, or the source file has been moved or is inaccessible), the debugger displays an assembly listing for the function if the target application is stopped.

A status panel in the lower frame of the browser window tells you whether the debugger was able to map the source code to the code on the target machine. The message “Targeted” means that the mapping succeeded. The message “Not Targeted” means that it did not. In this case, check that you have launched the target application and that it is not currently executing.

At the lower-left corner of the browser window, a pop-up menu allows you to switch between source and assembly views of the selected function. Notice also the area to the left of the vertical dotted line in the code view; this is the breakpoint column. Clicking in this column sets a simple breakpoint on the corresponding source code statement or instruction. See “Setting and Clearing Breakpoints” beginning on page 3-9 for more information.

Figure 3-2 shows a sample browser window. If you select multiple symbolic information files, the debugger opens a browser window for each file.

Figure 3-2 A sample browser window



The browser window shown in Figure 3-2 is an example of the typical display you get when you enter the debugger using the `DebugStr` routine. The source file that appears in a box (in the upper-left pane) indicates that you selected it and that this pane isn’t currently active. Your most recent selection, the routine containing the call to `DebugStr`, is highlighted. The source code for the function is shown in the code view.

Displaying Multiple Code Views

As mentioned in the previous section, you can switch from source to assembly views in the browser's code view by selecting Source or Assembler from the pop-up menu. There may be times, however, when you want to view assembly and source code listings for the same function simultaneously, or display multiple views of the same source code. To do this follow these steps:


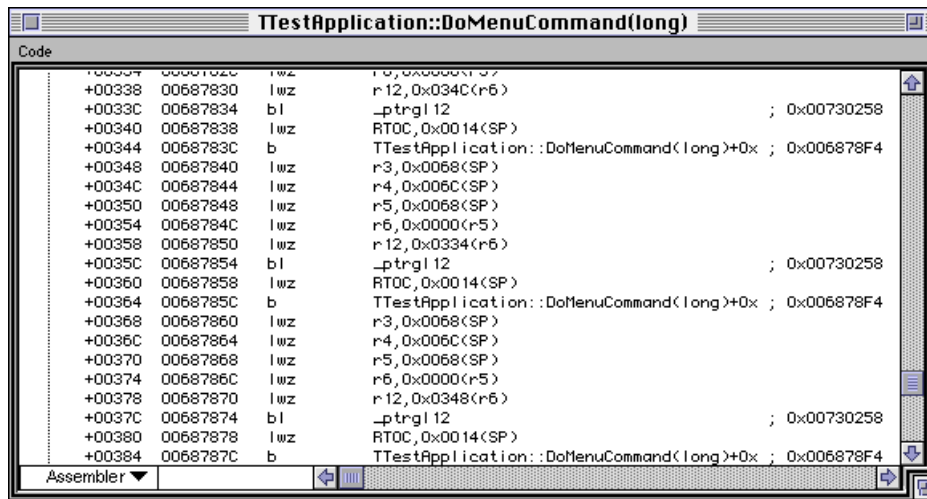
1. Place the cursor in the code view of the browser window.
2. Hold down the Option key. The cursor changes into a small window icon. 
3. Click in the pane, then drag and release to create a code view window displaying the same function.
4. Select either Source or Assembler from the pop-up menu in the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

Figure 3-3 shows the assembler view of the code shown in Figure 3-2.

Figure 3-3 An alternative code view



Finding Code Quickly

You can locate a routine quickly and display it in its own window with commands in the Views menu. Choose Find Code For from the Views menu, and a dialog box like the one shown in Figure 3-4 is displayed. Enter the name of the routine you want to find in the text box and click OK. You must give the full class name for C++ methods.

Figure 3-4 The Find Code For dialog box

Another way of finding code and displaying it is to first select a routine name in the browser window. This routine name then appears in the Views menu as the item Find Code For plus the name of the selected routine. For example, if the following source code is shown in the browser window and you select `doSomething`, the menu item becomes Find Code For `doSomething`. If you choose this item, the debugger finds the source code for `doSomething` and displays it in its own window.

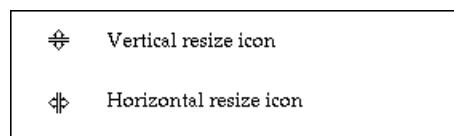
```
void doNothing (short itemNum)
{
    if (itemNum==kCallMyRoutine)
        doSomething();
}
```

Manipulating Windows

The Views menu allows you to create windows that display specific areas of memory, including the stack, your application's global variables, the machine's registers, and memory segments.

Once you have opened several windows, you can bring a window forward and make it active either by clicking in the window or by choosing the window's name from the Window menu. You close a window either by clicking its close box or by choosing Close (Command-W) from the File menu.

You can resize the panes in some of the debugger windows, for example the browser and stack windows, by placing the cursor on the split bar (the triple lines separating the panes) and pressing to display the resize icon. Figure 3-5 shows the vertical and horizontal resize icons. When a resize icon is displayed, drag to the left or right (or up or down) to resize the panes.

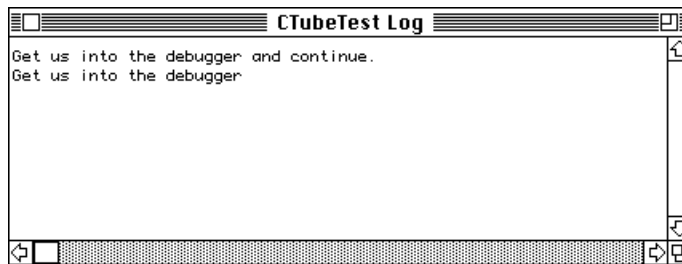
Figure 3-5 Resize icons

Using the Debugger

The Log Window

The debugger creates a log window and writes the output of `DebugStr` calls to this window. Figure 3-6 shows an example of a log window for the sample application `CTubeTest`. You can save the contents of this window to an MPW text file by choosing `Save Log As` from the `File` menu.

Figure 3-6 The log window



Printing the Contents of a Window

You can print the contents of any window by choosing `Print Window` from the `File` menu. Whether the debugger prints just the visible contents of the window or the entire scrollable region of the window depends on the type of window being printed. For the memory and instruction windows, which scroll over the entire address space, only the visible portion of the window is printed. For code views, all the source code or assembly-language code for the selected function is printed.

Taking a Snapshot of the Active Window

Choose `Snapshot Active Window` from the `Extras` menu to save a snapshot of the current frontmost window. When you expect to change the state of the frontmost window, you can save a snapshot to use for comparison at a later time.

Remember Window Position








Choose `Remember Window Position` from the `Edit` menu to save the location and size of the active window in the preferences file. The window is saved by window type only, not by instantiation. When you open multiple windows of the same type, they come up in a staggered position. This option also remembers the size and position of any panes within windows that have them, such as the browser window and the stack window.

Setting and Clearing Breakpoints

To examine the state of your process during execution, you must be able to halt execution and examine the parts of memory affected by this process. The Macintosh Debugger provides a variety of ways of halting program execution, including the ability to set simple breakpoints, conditional breakpoints, and counting breakpoints. This section explains how you set breakpoints from the code view of the browser window. The next section, “Controlling Program Execution” (page 3-15) explains how you resume program execution after you have set a breakpoint. “Looking at Memory” beginning on page 3-18 describes the regions of memory you can examine when the executing process is halted. (You can also set breakpoints from the instruction display window. See “Displaying PowerPC Instructions” on page 3-21 for additional information.)

The debugger allows you to set five kinds of breakpoints, each designated by a special breakpoint icon. Table 3-1 describes the breakpoint icons.

Table 3-1 Debugger breakpoint icons

Name	Icon	Use
Simple		Program execution stops every time this breakpoint is encountered
One shot		Program execution stops when this breakpoint is encountered. The breakpoint is then automatically removed.
Counting		Program execution stops the <i>n</i> th time the breakpoint is encountered. You specify the value of <i>n</i> .
Conditional		Program execution stops when a condition you specify evaluates to true.
Performance		Performance measurement is turned on or off when these breakpoints are encountered. A separate icon is used to mark on and off operation. The debugger stops at performance breakpoints if the hollow square is checked; otherwise performance measurement is just turned on and off and then continues.
		
		

Note

If you set a breakpoint in code that is not targeted, the breakpoint icon will have an “unhappy face” in it. ♦

The hollow diamond icons displayed in the breakpoint column indicate where it is possible to set breakpoints. Once you have set a breakpoint, its icon is shown in the

Using the Debugger

breakpoint column of the browser's code view or of the instruction window. The breakpoint column is the narrow space to the left of the vertical dotted line. Figure 3-1 on page 3-4 shows the location of the breakpoint column in the browser window.

You can set a breakpoint at any statement or instruction:

- To set a breakpoint on a source code statement, set it from the browser's source code view.
- To set a breakpoint on an instruction, set it from the browser's assembler code view.

You can set a simple breakpoint by clicking in the breakpoint column. You can set a one-shot breakpoint by pressing Command-Option while clicking in the breakpoint column. To set a counting, conditional, or performance breakpoint, press Option while clicking in the breakpoint column. A dialog box appears from which you can select the type of breakpoint you want.

To remove any breakpoint, place the cursor on top of the breakpoint icon and click once. To remove all breakpoints, choose the Clear All Breakpoints item from the Control menu.

The following sections give more details on setting the various kinds of breakpoints.

Simple Breakpoints

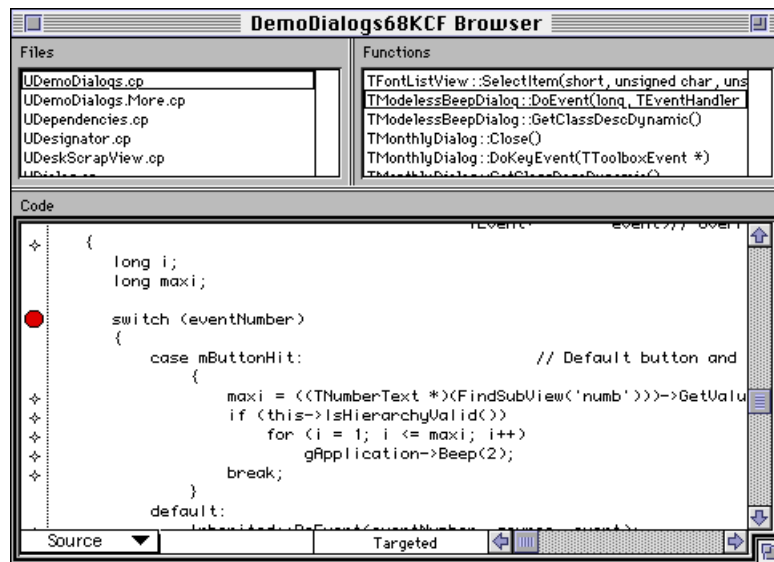
A **simple breakpoint** stops program execution every time it is encountered. To set a simple breakpoint

- 1. Make the browser's code view or the instruction window active.**
- 2. Place the cursor in the breakpoint column.**

When it is in the breakpoint column (to the left of the dotted line), the cursor changes to an octagon (stop sign shape) with an arrow in it.

- 3. Place the octagon next to the desired source statement or instruction and click once.**
This places an octagon next to the line of code where the breakpoint is set. (On a color monitor the octagon is red with a black frame.)

Figure 3-7 shows the browser window after a simple breakpoint has been set.

Figure 3-7 Setting a simple breakpoint

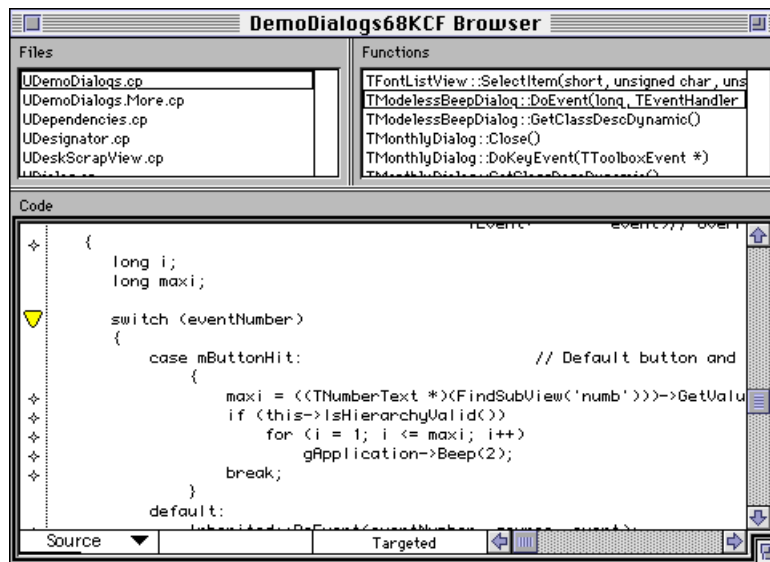
One-Shot Breakpoints

A **one-shot breakpoint**, or temporary breakpoint, allows you to set a breakpoint and resume execution of the target application with a single mouse click. To set a one-shot breakpoint

1. Make the browser's code view or instruction window active.
2. Move the cursor to the breakpoint column (left of the dotted line in the code view) next to the statement or instruction where you want to place the breakpoint. Press **Command-Option** and click.

This places a triangle (yield sign shape) next to the line of code where the breakpoint has been set, resumes execution of the target application, and stops at the one-shot breakpoint. The breakpoint icon is then automatically removed. (On a color monitor the triangle is yellow with a black frame.)

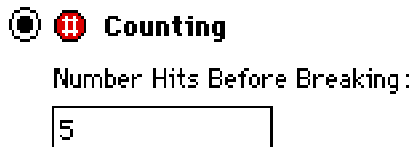
Figure 3-8 shows the browser window after a one-shot breakpoint has been set.

Figure 3-8 Setting a one-shot breakpoint

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

1. Make the browser's code view or instruction window active.
2. Move the cursor to the breakpoint column, next to the statement or instruction where you want to place the breakpoint. Press the Option key and click.
A dialog box is displayed, allowing you to select a type of breakpoint. Figure 3-9 shows the control you use to specify a counting breakpoint.
3. Click the radio button for the counting breakpoint and enter a number in the text box below it.
This places the counting breakpoint icon next to the line of code where the breakpoint has been set.

Figure 3-9 Selecting a counting breakpoint

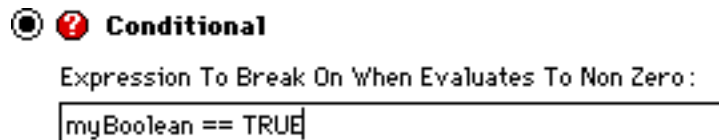
Conditional Breakpoints

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

1. **Make the browser's code view or instruction window active.**
2. **Move the cursor to the breakpoint column, next to the statement or instruction where you want to place the breakpoint. Press the Option key and click.**
A dialog box is displayed, allowing you to select a type of breakpoint. Figure 3-10 shows the control you use to specify a conditional breakpoint.
3. **Click the radio button for the conditional breakpoint and enter an expression in the text box below.**

This places the conditional breakpoint icon next to the line of code where the breakpoint has been set. Expressions for conditional breakpoints in instruction windows cannot use variables or symbolic types, just registers, addresses, and numeric constants. For additional information about expression grammar, see Appendix A, “Expression Evaluation.”

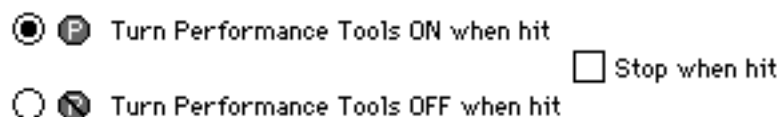
Figure 3-10 Selecting a conditional breakpoint



Performance Breakpoints

A **performance breakpoint** allows you to automatically start or stop performance measurement for selected blocks of code. A dialog box like the one shown in Figure 3-11 is displayed. For additional information, see the section “Measuring Selected Routines” on page 4-8.

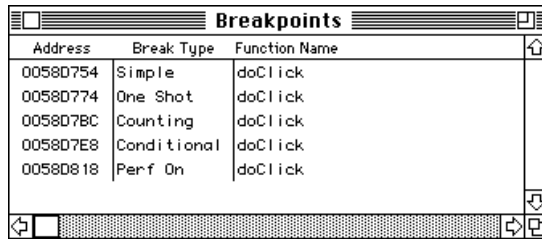
Figure 3-11 Setting a performance breakpoint



Displaying Breakpoints

You can display a list of all currently set breakpoints by choosing Show Breakpoints List from the Control menu. The debugger displays a window similar to the one shown in Figure 3-12.

Figure 3-12 Displaying breakpoints



Address	Break Type	Function Name
0058D754	Simple	doClick
0058D774	One Shot	doClick
0058D7BC	Counting	doClick
0058D7E8	Conditional	doClick
0058D818	Perf On	doClick

The display shows the function name, the function's starting address, and the type of breakpoint set at that address.

- If you double-click the function name, the source view of the function is displayed in the browser window.
If the debugger displays the function name "???" for a breakpoint, it indicates that the debugger could not find symbolic information or embedded names mapped to the address where the break has been set. You cannot display code for this function.
- If you double-click the address, the disassembly for the function is shown in the instruction window.
If the address column is blank for a breakpoint, the breakpoint has not actually been set on the target machine yet. This happens if you set a breakpoint in the browser window when the target process is not targeted or if you open a symbolic information file before launching the target application.

You can delete a breakpoint by selecting the breakpoint in the breakpoints window and pressing Delete. The debugger removes the breakpoint from your application and from the breakpoints window.

Setting Breakpoints From Your Source Code

You can use the debugger to set breakpoints, or you can call two system routines, `Debugger` and `DebugStr`, from your source program to set breakpoints.

- When the debugger encounters the `Debugger` routine, it stops the program. The program counter points to the next instruction to be executed.

The syntax of the `Debugger` routine is

```
Debugger ( ) ;
```

Using the Debugger

- When the debugger encounters a `DebugStr` routine, it stops the program and writes the output of the routine, a string, to the log window. The program counter points to the next instruction to be executed.

The following is an example of a call to `DebugStr`:

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

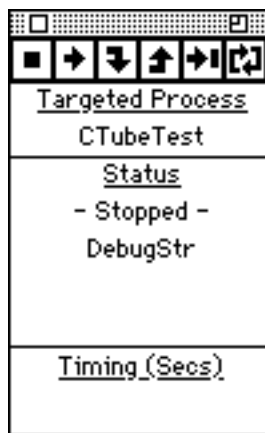
Disabling Calls to Debugger or DebugStr

You can disable calls to the Debugger or `DebugStr` routines by selecting the item `Stop For DebugStrs` from the Extras menu.

Controlling Program Execution

When your program is stopped, you can resume execution by choosing an item from the Control menu or by clicking an icon in the control palette. The control palette is a floating window that is opened at startup; if you have closed it, you can open it again by selecting `Show Control Palette` from the Control menu. Figure 3-13 shows the control palette.

Figure 3-13 The control palette



In addition to providing program execution controls, the palette also displays the following status information:

- **Targeted Process:** The name of the currently targeted process.
- **Status:** The program is either launching, running, or stopped.

Using the Debugger




- Cause of break if the program is stopped: This might be a user break (such as `DebugStr`), breakpoint, or other exception.
- 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.

The sample control palette in Figure 3-13 indicates that the target program `CTubeTest` is stopped, that the reason for the break is the execution of the `DebugStr` routine, and that this is the first break recorded; hence there is no timing information.

To shrink the control palette so that it shows only the control icons, click the zoom box in the upper-right corner.




Table 3-2 shows the correspondence between Control menu items and control palette icons, and describes the effect of each program control command. These commands operate at the source level if the active window is displaying a source view or at the machine level if the active window is displaying an assembler view. If neither, the debugger uses the last view.

Table 3-2 Program control commands

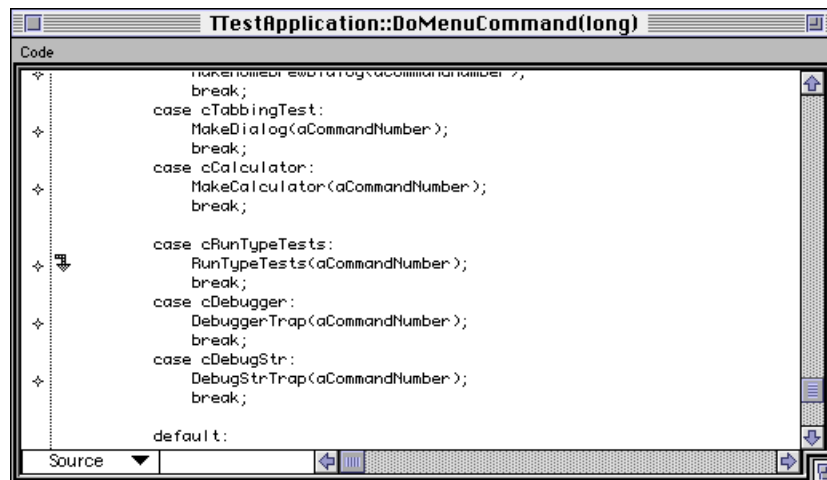
Control menu item	Keyboard equivalent	Palette icon	Effect
Run	Command-R		Resumes execution of the target application.
Stop	(none)		Halts execution of the target application and allows you to set breakpoints and resume controlled execution of the target. (The Stop menu item is supported only for low-level debugging with Power Mac Debugger.)
Propagate Exception	Command-Option-R	(none)	Propagates an exception to your application's exception handler. Choose this item following an exception if your application includes an appropriate exception handler.
Step	Command-S		<p>Steps through the source code, stepping over procedure calls. Stepping takes place at the source level if a source code view is the front window and at assembly level if an assembly view is frontmost. If you bring up another window, for example a memory window or a stack window, stepping takes place at the level of the last frontmost (source- or assembly-view) window.</p> <p>If you have a browser window as the front window and step into code that is in another browser, that window is brought to the front. If you step into code that has no associated source code, the debugger brings up an instruction window to display the PC.</p>

Using the Debugger

Table 3-2 Program control commands (continued)

Control menu item	Keyboard equivalent	Palette icon	Effect
Step Into	Command-I		Steps into procedure calls.
Step Out	Command-T		Steps out of the current (called) routine and stops at the next instruction in the calling routine.
Animate	(none)		Continuously executes the next Step, Step Into, or Step Out command that you choose. The program stops when it reaches a breakpoint. You can select Animate again to disable this mode even while the target is executing. The menu item is checked when this mode is enabled.

When the program is stopped, the current PC is displayed as an arrow to the left of the current statement. This arrow is a solid color in the routine of the topmost frame; the arrow used in previous stack frames (to show what the PC is for those frames) is filled with a gray pattern and points down to denote that the PC currently points to a statement or instruction within the subroutine. Figure 3-14 shows an example of this.

Figure 3-14 The PC in subroutine marker

The Control menu offers a few additional commands for controlling program execution that aren't available in the control palette. Commands to help you manually target a running process, launch an application, and step to a branch are among those described next.

Target “”

Choose Target “*Process Name*” from the Control menu to manually target a running process previously selected through the Show Tasks command in the Views menu. You should see one or two progress meters while the debugger reads the process information. Then a registers window (or stack window, depending on the preferences setting) is opened for the newly targeted process. The initial contents of the window are not valid, because the process is not stopped.

Untarget “”

Choose Untarget “*Process Name*” from the Control menu to untarget a previously targeted process.

Launch

Choose Launch from the Control menu to launch an application and have it stop before executing `main`. This command is helpful for debugging initialization code. The Launch command only works for one-machine debugging; to break on launch using two-machine mode, hold down the Control key on the remote machine as you double-click the application.

Step to Branch

Choose Step To Branch from the Control menu (or press Command-B) to cause the debugger to automatically step over all instructions until it reaches any branch instruction, at which point it stops. This command can be used at source or assembly level, but makes most sense when used at assembly level.

Step to Branch Taken

Step To Branch Taken has the same effect as Step To Branch, except that it stops only if the branch is actually going to be taken. This command can be used at source or assembly level, but makes most sense when used at assembly level.

Looking at Memory

You can use the debugger to examine different sections of memory. This section explains how you select these displays from the Views menu, how you interpret the information displayed, and how, in some instances, you can change memory values.

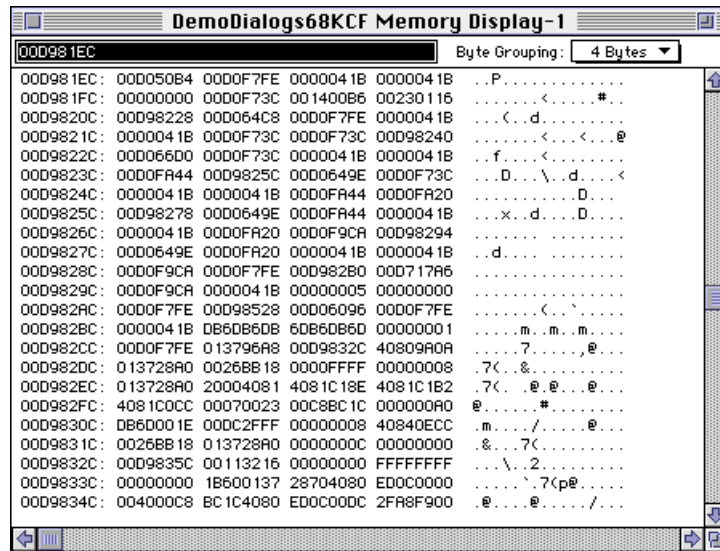
To display memory, choose Show Memory from the Views menu. Figure 3-15 shows the memory display window, which views the full memory range.

Using the Debugger

Note

For 68K Mac Debugger, the default starting address of the memory display is register A7; for Power Mac Debugger, the default starting address of the memory display is at the stack pointer. ♦

Figure 3-15 The memory display using 4-byte alignment



When you open a memory display window, the debugger tries to convert the current selection in the topmost window to a hexadecimal address and, if it succeeds, it starts the memory dump from that address. Otherwise, the debugger starts at the stack pointer.

To scroll to a particular address, use the scroll bar or enter the address in the box in the upper-left corner of the window and press Tab or Enter.

You can change memory 4 bytes, 2 bytes, or 1 byte at a time:

1. Click the value you want to change.

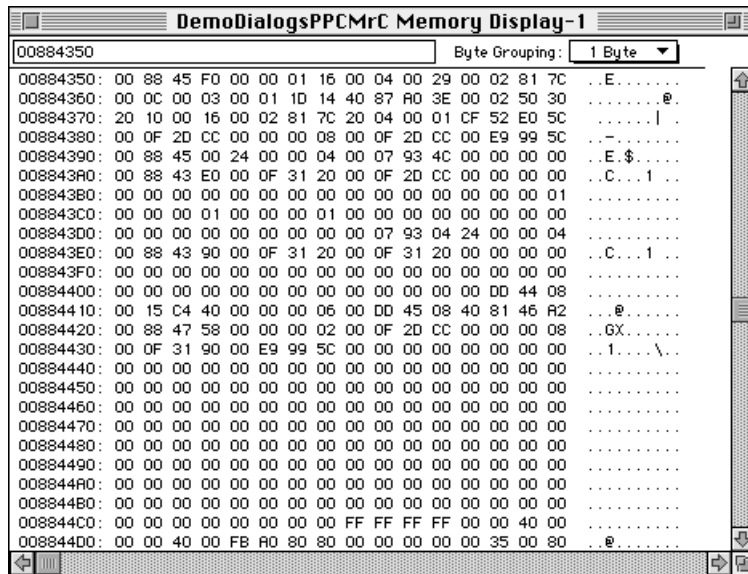
2. Edit the value.

If you change your mind, press the Escape key to stop editing without changing the value stored in memory.

3. Press the Enter or Tab key to make the change.

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

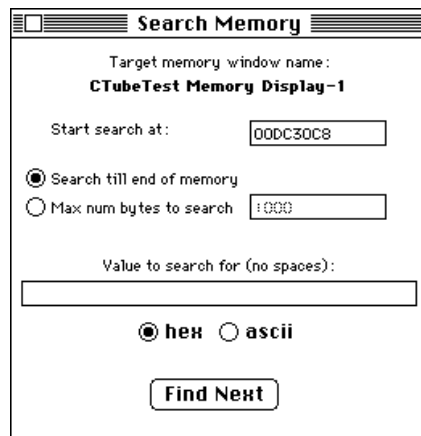
You can change the grouping used to display values in memory by selecting an alternative value from the Byte Grouping pop-up menu. Figure 3-16 shows memory displayed using 1-byte alignment.

Figure 3-16 The memory display using 1-byte alignment

Memory is grouped as 4-byte values by default. To change the default memory-grouping value, choose General Preferences from the Edit menu and select an alternative value.

Searching Memory

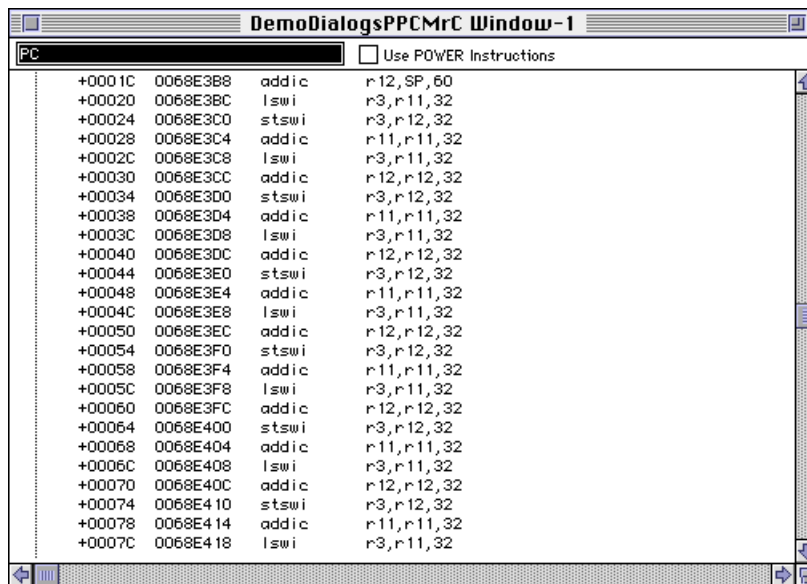
Choose Search Memory from the Extras menu to display the Search Memory window, shown in Figure 3-17. The Search Memory menu item is enabled only when a Memory Window is the frontmost active window. The debugger supports searching for ASCII or hexadecimal values. The Search Memory window is tied to the specific instantiation of a memory window and closes when that window closes.

Figure 3-17 Search Memory window

The following two sections explain how you display sections of memory disassembled with the PowerPC disassembler or with the 680x0 disassembler. For additional information about memory organization on Power Macintosh computers, please see Chapter 5, "Low-Level Debugging."

Displaying PowerPC Instructions

To display a section of memory disassembled with the PowerPC disassembler, choose Show Instructions from the Views menu or press Command-D. The debugger opens an instruction display window like the one shown in Figure 3-18.

Figure 3-18 The PowerPC instruction display window

When you choose this item, the debugger attempts to convert the selection in the toplevel window to an address. If it succeeds, disassembly begins from that address. Otherwise, disassembly begins at the program counter. The disassembly address is shown in the box at the upper-left corner of the display. To change the address where disassembly begins, enter a new value in the box and press the Tab or Enter key to display the new disassembly.

You can view instructions using either the POWER instruction set mnemonics or the PowerPC mnemonics. The initial default setting is to display instructions using PowerPC mnemonics.

- To view instructions using POWER mnemonics, click the Use POWER Instructions checkbox. This affects the current session.
- To change the default setting to use POWER mnemonics, choose General Preferences from the Edit menu and click the appropriate checkbox.

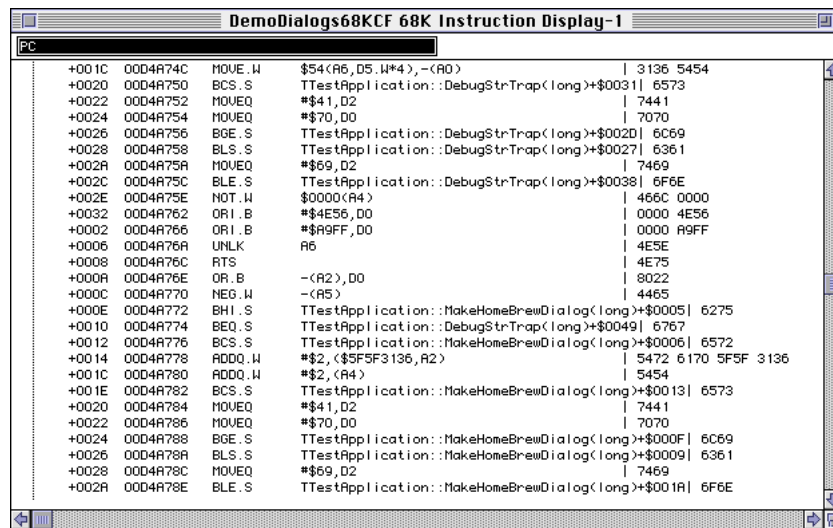
You can set breakpoints in the instruction display view by placing the cursor in the breakpoint column and clicking to the left of the desired instruction. You can clear breakpoints or resume program execution in the usual way.

For additional information about PowerPC instructions, see *Assembler for Macintosh With PowerPC* or Motorola's *PowerPC 601 RISC Microprocessor User's Manual*.

Displaying 680x0 Instructions

To display a section of memory disassembled with the 680x0 disassembler, choose Show 68K Instructions from the Views menu or press Command-8. The debugger opens an instruction display window like the one shown in Figure 3-18.

Using the Debugger

Figure 3-19 The 680x0 instruction display window

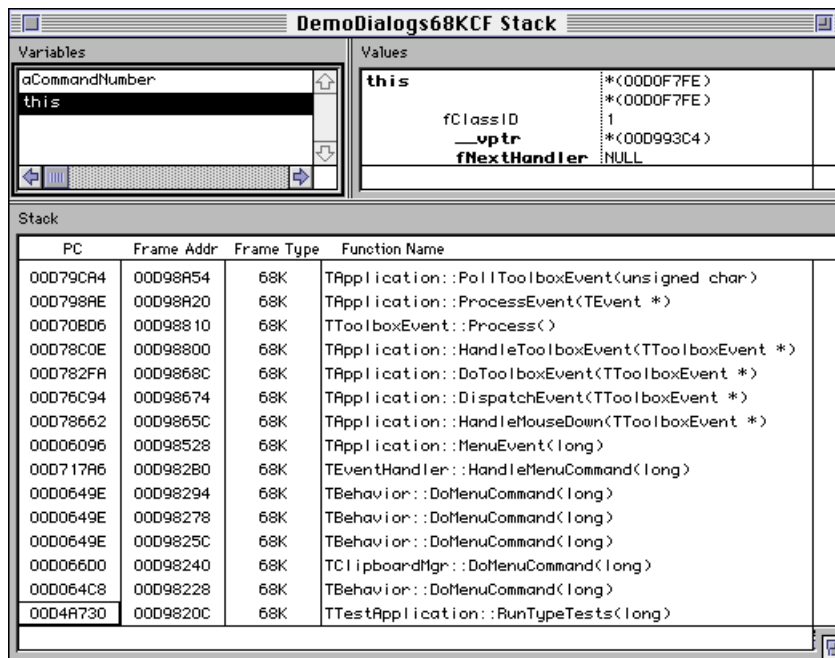
When you choose this item, the debugger attempts to convert the selection in the toptmost window to an address. If it succeeds, disassembly begins from that address. Otherwise, disassembly begins at the program counter. The disassembly address is shown in the box at the upper-left corner of the display. To change the address where disassembly begins, enter a new value in the box and press the Tab or Enter key to display the new disassembly.

Note

The 680x0 instruction display window for Power Mac Debugger is provided for reference only. It does not display the current PC nor can you set breakpoints in this view. The validity of the disassembled instructions depends on your specifying a valid starting address for disassembly. ♦

The Stack

When you choose Show Stack Crawl (Command-J) from the Views menu, the debugger displays a three-pane window like the one shown in Figure 3-20.

Figure 3-20 The stack display

You can resize the top panes by placing the cursor on the split lines and pressing to display the horizontal resize icon. When the icon is displayed, drag to the left or right to resize the panes.

The lower pane is a stack crawl. For each stack frame, it displays the program counter, the stack pointer, the type of frame, and the name of the routine (if the name is known.) If the name is not known, it is indicated by a series of question marks (????). The last stack frame shown is the one at the top of stack—that is, it is the frame for the current function. Remember that the stack grows toward low memory.

If you select a stack frame in the stack crawl window, the debugger displays the local variables for that frame in the upper-left pane. If you select one of these variables, the debugger displays its value in the upper-right pane. If the variable is a structured data type, the debugger displays the value of each of its fields. Special notation is used to designate addresses, pointers, and handles in the upper-right pane.

- a value in parentheses specifies an address
- an address preceded by an asterisk (*) specifies a pointer

You can access an instruction, memory, or browser display directly from the stack window by double-clicking selected items in the bottom pane:

- If you double-click a PC address, the debugger displays an instruction window, with disassembly beginning at the PC address.
- If you double-click a frame address, the debugger displays a memory window, with the memory dump beginning at the frame address.

Using the Debugger

- If you double-click a function name, the debugger displays a browser window or an instruction window.

To evaluate a local variable in a recursive routine, select the frame that you are interested in. The local variables for that frame are displayed in the upper-left pane as usual.

For additional information about the structure of stack frames, please see Chapter 5, “Low-Level Debugging.”

Registers

The debugger allows you to display and change the values of the PowerPC and 680x0 general-purpose and floating-point registers.

PowerPC General-Purpose Registers

To display the PowerPC general-purpose registers, choose the Show Registers item from the Views menu. A registers window like the one shown in Figure 3-21 is displayed.

Figure 3-21 The PowerPC registers window

Muslin Registers																	
		CR0		CR1		CR2		CR3		CR4		CR5		CR6		CR7	
PC		<u>0058D3C8</u>		CR		<u>0 1 0 0 0 0 1 0</u>		<u>0000</u>		<u>0000</u>		<u>0000</u>		<u>0000</u>		<u>0100 1000</u>	
LR		<u>0058D3C8</u>		< > = 0 X E U 0													
CTR		<u>409E0DF0</u>		S O C		Compare		Count									
		XER		<u>0 0 0</u>		<u>00</u>		<u>00</u>									
R 00		<u>000CB648</u>		R 08		<u>6808D434</u>		R 16		<u>00000000</u>		R 24		<u>00000000</u>			
SP		<u>005CA230</u>		R 09		<u>00000010</u>		R 17		<u>00000000</u>		R 25		<u>001388E2</u>			
TOC		<u>00011240</u>		R 10		<u>00020B20</u>		R 18		<u>00000000</u>		R 26		<u>00000001</u>			
R 03		<u>0059403F</u>		R 11		<u>00000000</u>		R 19		<u>00000000</u>		R 27		<u>00000001</u>			
R 04		<u>00033022</u>		R 12		<u>0058D3C8</u>		R 20		<u>00000000</u>		R 28		<u>005CA448</u>			
R 05		<u>FFFFA1AD</u>		R 13		<u>00000000</u>		R 21		<u>00000000</u>		R 29		<u>0059853C</u>			
R 06		<u>636E666E</u>		R 14		<u>00000000</u>		R 22		<u>00000000</u>		R 30		<u>00598530</u>			
R 07		<u>005CA0E0</u>		R 15		<u>00000000</u>		R 23		<u>00000000</u>		R 31		<u>00594038</u>			

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. You can change the value of a register by clicking its value, entering a new value, and pressing the Enter or Tab key. 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 on a bit to toggle its value.

Using the Debugger

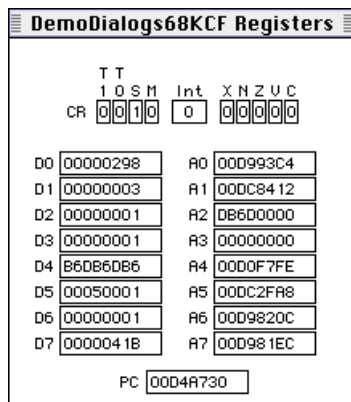
Note

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). ♦

680x0 General-Purpose Registers

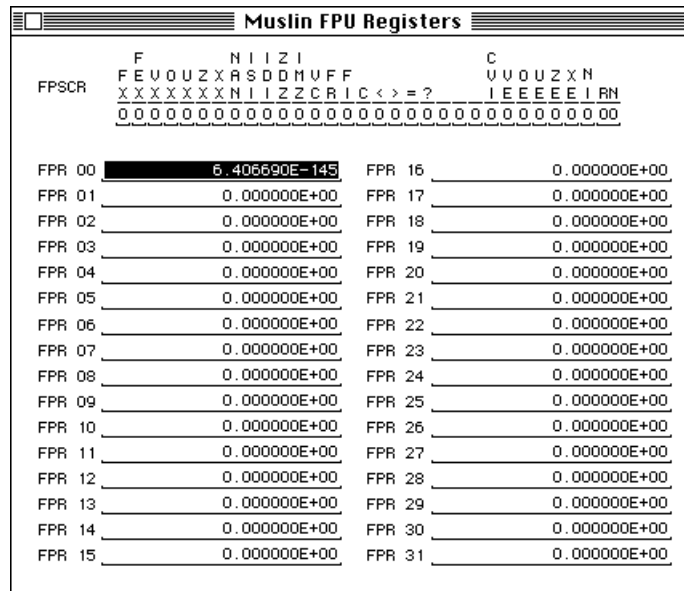
To display the 680x0 general-purpose registers, choose the Show Registers item from the Views menu. A registers window like the one shown in Figure 3-22 is displayed.

Figure 3-22 The 680x0 registers window



PowerPC Floating-Point Registers

To display the PowerPC floating-point registers, choose Show FPU Registers from the Views menu. The debugger displays a window like the one shown in Figure 3-23.

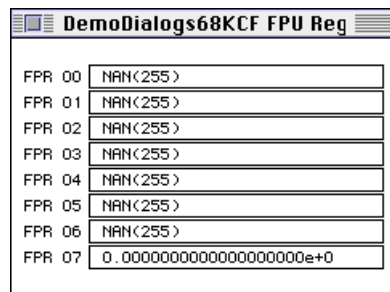
Figure 3-23 The PowerPC floating-point registers

The floating-point registers window displays 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 the Enter or Tab key. To undo a change, choose Undo from the Edit menu.
- The FPSCR value is displayed in binary. Click on a bit to toggle its value.

680x0 Floating-Point Registers

To display the 680x0 floating-point registers, choose Show FPU Registers from the Views menu. The debugger displays a window like the one shown in Figure 3-24.

Figure 3-24 The 680x0 floating-point registers

Displaying the Value of Variables

You can use items in the Evaluate menu to display the value of

- a symbolic name or address
- an expression
- "this" in C++

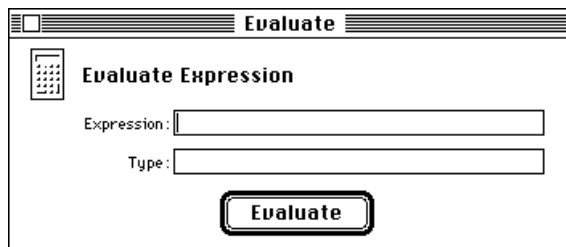
Additional items in the Evaluate menu allow you to view a selected value in a variety of formats.

Names, Addresses, or Expressions

You can use one of two methods to display the value of a name, address, or expression: choosing either Show Evaluation Window or Evaluate "" from the Evaluate menu.

If you choose Evaluate... from the Evaluate menu, the debugger displays a dialog box like the one shown in Figure 3-25.

Figure 3-25 The Evaluate dialog box



Enter the variable name, address, or expression in the Expression text field.

- If you specify an address, you must also enter a string in the Type text field that specifies the data type used to interpret the contents in memory; for example: "short" or "windowrecord*".
- If you specify a name, the Type entry is optional.
- If you specify an expression, the Type entry is optional.

Here are some examples of valid expressions:

```
&a[0] != ptrs[0]
0xabc + 1234L - Δr1 + Δfp0      /* Δr1 specifies GPR 1 */
                                /* Δfp0 specifies FPR 0 */

sizeof("\abcdef\")
$abc + #12L + fp0              /* $ specifies hexadecimal */
```

Using the Debugger

```

/* # specifies decimal */
(b << 4) >> 8
>(*Handle)->substruct)
'\\n' + '\\t' + '\\v' + '\\0123' - '\\x1fa' - 'abc'

```

For additional information about the evaluation of expressions, see Appendix A.

To display the value of a variable or expression in any code view, select the variable and choose Evaluate "" from the Evaluate menu. The text of the menu item is completed with the name of your selection. For example, if you select the variable `MyCounters`, the Evaluate "" item in the Evaluate menu reads Evaluate `MyCounters`.

To change the display format of a selected variable in any code view, choose one of the following items from the Evaluate menu:

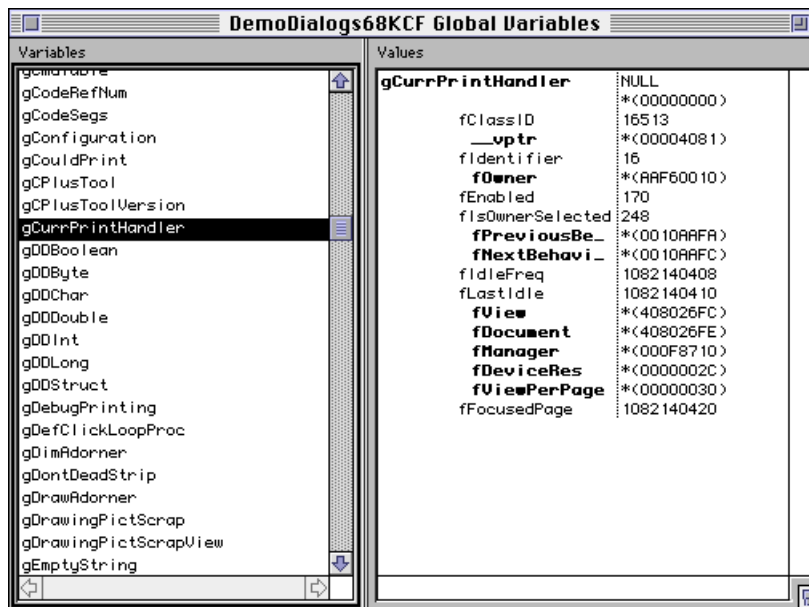
Item	Effect
View as Default Type	Display the selected value using the appropriate default format. The default format for numbers is decimal; for unsigned longs, the default type depends on the current preference settings (set in dialog box you get by choosing General Preferences from the Edit menu); the default format for strings is an array of chars.
View as Character	Display the selected value as a character.
View as Decimal	Display the selected value as a 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 Str255	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.

The Value of “this”

If you select Evaluate “this” from the Evaluate menu and the target program is stopped in a C++ method, the debugger displays the value of the instance of the current class.

Global Variables

To display the names and values of your program’s global variables, choose Show Globals from the Views menu or press Command-L. The debugger opens a two-pane window like the one shown in Figure 3-26.

Figure 3-26 Displaying global variables

The left pane of the global variables window lists your program's global variables. If multiple symbolic files are open, the source file name will also be shown to the right of the global variable name. To display the value of a variable, select the variable. Its value is shown in the right pane. If the variable is a structured data type, the value of each field is also shown in the right pane.

Special notation is used to designate addresses, pointers, and handles in the right pane.

- a value in parentheses specifies an address
- an address preceded by an asterisk (*) specifies a pointer

You can change the value of a global variable by selecting its value in the right pane and editing it. Press Tab or Enter to make the change.

You can resize the panes by placing the cursor on the vertical lines separating the panes (split lines) and pressing to display the horizontal resize icon. Then drag to the left or right to resize the panes.

Handling Fragments

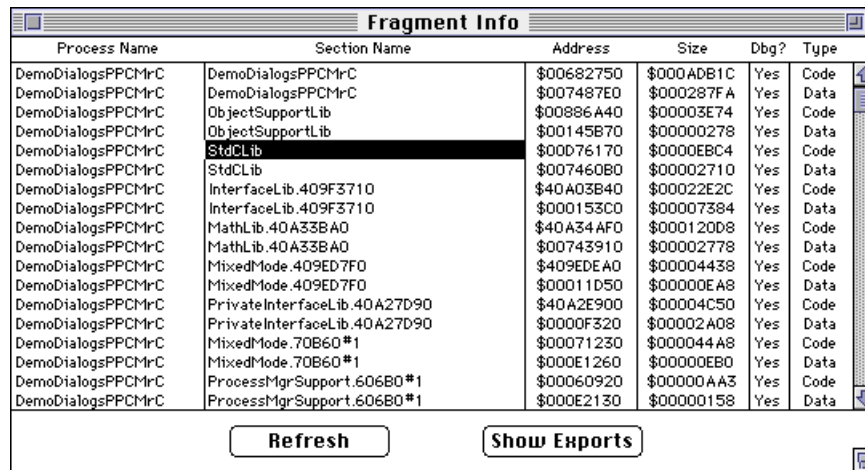
Several debugger options make it easier for you to handle fragments.

Using the Debugger

Show Fragment Info

Choose Show Fragment Info from the Views menu to bring up a window that displays a list of all fragments on the target machine, as shown in Figure 3-27. The list includes all fragments on the target machine, not just those being debugged. Use the Refresh button to refetch all the data from the nub. The Refresh button is useful because this window is not automatically updated when fragments are created or deleted on the target machine.

Figure 3-27 Fragment Info window



Process Name	Section Name	Address	Size	Dbg?	Type
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$00682750	\$000ADB1C	Yes	Code
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$007487E0	\$000287FA	Yes	Data
DemoDialogsPPCMrC	ObjectSupportLib	\$00886A40	\$00003E74	Yes	Code
DemoDialogsPPCMrC	ObjectSupportLib	\$00145B70	\$00000278	Yes	Data
DemoDialogsPPCMrC	StdCLib	\$00D76170	\$0000EBC4	Yes	Code
DemoDialogsPPCMrC	StdCLib	\$007460B0	\$00002710	Yes	Data
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$40A03B40	\$00022E2C	Yes	Code
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$000153C0	\$00007384	Yes	Data
DemoDialogsPPCMrC	MathLib.40A33BA0	\$40A34AF0	\$000120D8	Yes	Code
DemoDialogsPPCMrC	MathLib.40A33BA0	\$00743910	\$00002778	Yes	Data
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$409EDEA0	\$00004438	Yes	Code
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$00011D50	\$00000EA8	Yes	Data
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$40A2E900	\$00004C50	Yes	Code
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$0000F320	\$00002A08	Yes	Data
DemoDialogsPPCMrC	MixedMode.70B60#1	\$00071230	\$000044A8	Yes	Code
DemoDialogsPPCMrC	MixedMode.70B60#1	\$000E1260	\$00000EB0	Yes	Data
DemoDialogsPPCMrC	ProcessMgrSupport.606B0#1	\$00060920	\$00000AA3	Yes	Code
DemoDialogsPPCMrC	ProcessMgrSupport.606B0#1	\$000E2130	\$00000158	Yes	Data

Refresh Show Exports

The Show Exports button brings up a window that lists all the exports for the selected item, as shown in Figure 3-28. (Show Exports is not available for 68K Mac Debugger.) You can sort the data in the Show Exports window in one of three ways: by symbol name, by address, or by symbol type. To sort the list, just click the column title to be used as the sort key. For example, click Symbol Name to sort by name.

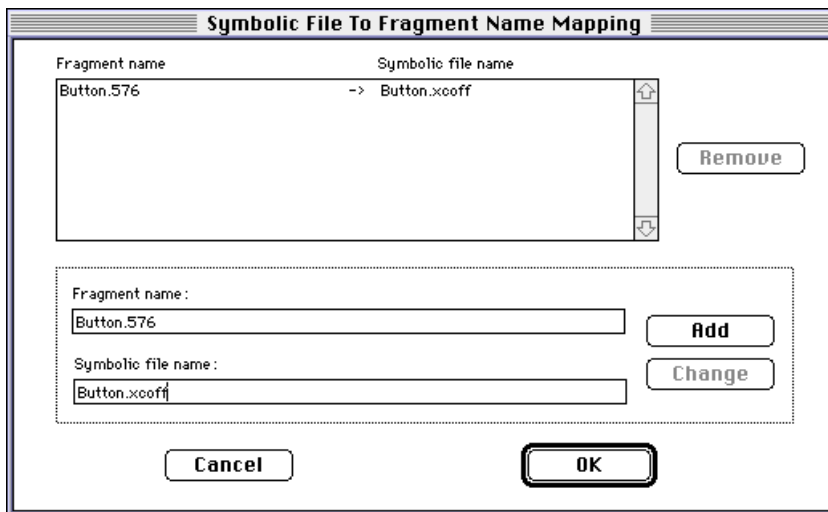
If you double-click a row that displays a Code Vector symbol type, then the debugger opens an instruction window at the dereferenced address for the symbol. If the row displays a data symbol, then you can double-click it to open a memory window at the specified address.

Figure 3-28 Show Exports window

Symbol Name	Address	Symbol Type
abort	\$00746CA4	Code Vector
abs	\$00746518	Code Vector
access	\$00746B54	Code Vector
asctime	\$00746524	Code Vector
atexit	\$00746FE0	Code Vector
atof	\$00746530	Code Vector
atoi	\$0074653C	Code Vector
atol	\$00746548	Code Vector
binhex	\$00746554	Code Vector
bsearch	\$00746560	Code Vector
calloc	\$0074656C	Code Vector
clearerr	\$00746998	Code Vector
clock	\$00746578	Code Vector
close	\$00746B60	Code Vector
ConvertTheString	\$00746830	Code Vector

Symbolic Mapping Preferences

Choose Symbolic Mapping Preferences from the Edit menu when you need to inform the debugger of the name of the symbolic file to use for a particular fragment that you are debugging. The debugger displays a dialog box like the one shown in Figure 3-29.

Figure 3-29 Symbolic File to Fragment Name Mapping dialog box

For Power Mac Debugger, symbolic file mapping might be needed when you have more than one fragment in a PEF container. When the nub notifies the host of loads, it appends an offset to the end of the fragment name, for example, `Button.576`. You could use the Map Symbolics to Code command in the File menu each time to inform the debugger of the offset. The Symbolic Mapping Preferences command allows you to set this up once so that it is done automatically thereafter (assuming the offset does not change). The next

Using the Debugger

time you open `Button.xcoff`, the browser's name will be `Button.576` instead of just `Button`. When the fragment is loaded within the targeted application, it is now automatically mapped to the symbolics in the browser.

For 68K Mac Debugger, one case requiring symbolic file mapping occurs when you are debugging a fat shared library or drop-in. A fat shared library or drop-in is created with the `MergeFragment` tool, which places both the PowerPC and 680x0 fragments in the data fork of the shared library. To distinguish fragment names, the fragment that appears after the first fragment in the shared library has its offset appended to its name. For example, a shared library named `Button` is created by merging the PowerPC and 680x0 fragments with the following command:

```
MergeFragment -x Button.ppc Button.68k Button
```

This places the 680x0 fragment after the PowerPC fragment in the shared library called `Button`, and the 680x0 fragment name will be something similar to `'Button.576'`. In order to debug this fragment, you must explicitly tell the debugger to match the `Button.576` fragment to the symbolic file named `Button.NJ` by using the Symbolic Mapping Preferences menu item.

The Adaptive Sampling Profiler

Contents

About the Adaptive Sampling Profiler	4-3
Using the Adaptive Sampling Profiler	4-5
Enabling Performance Measurement	4-6
Starting a Profiling Session	4-6
Specifying a Sampling Rate	4-7
Collecting Performance Data	4-8
Measuring Selected Routines	4-8
Generating a Performance Report	4-9
How the ASP Presents Collected Data	4-9
The Summary View	4-11
The Statistics View	4-11
Editing the Performance Report	4-13
Saving and Printing a Performance Document	4-14
Evaluating Performance Data	4-15
How the Adaptive Sampling Profiler Gathers Data	4-15
Allocating Nodes	4-16
Splitting Buckets	4-17
Freeing Nodes	4-19
Registration Errors	4-19
ASP Overhead	4-19

The Adaptive Sampling Profiler

This chapter explains how you use the Adaptive Sampling Profiler (ASP) to measure how often sections of your code execute. The ASP is an integral part of the debugger; you enable the ASP by a selection from the debugger's Performance menu. Unlike other samplers, the ASP does not require you to modify your source code in order to take measurements. This means that you do not have to recompile your source to measure its performance. In addition, you can obtain a more reliable picture of your application's performance because you do not have to worry about paging behavior or memory use that is due to inserted sampling code.

This chapter begins with a brief overview of the ASP. Then it explains how you can

- collect performance data
- generate a report for the collected data
- focus on specific routines in your code
- measure the performance of shared libraries and other specialized code

The final section of this chapter, "How the Adaptive Sampling Profiler Gathers Data," explains the inner workings of the ASP in greater detail. You do not have to read this section to use the ASP, but reading it may improve your understanding of the performance results.

About the Adaptive Sampling Profiler

The **ASP (Adaptive Sampling Profiler)** is a sampling utility that allows you to measure how often sections of code execute. Like most other sampling tools, it collects performance data by allocating a number of counters and associating each counter with a discrete memory range. It then samples the program counter (PC) at regular intervals and increments the counter that corresponds to the region of memory containing the PC address. This counter is called a **bucket**. When the counter is incremented, the bucket is said to be **hit**. The frequency with which the PC is sampled is called the **sampling rate**. The default sampling rate value used by the ASP is 10 milliseconds (ms).

The ASP, unlike most sampling tools, does not associate the same size region of memory with each bucket. Instead, it dynamically adjusts the sizes of addressable code regions associated with certain buckets in order to give you the finest granularity for those routines or instructions that execute most often. The profiler is called *adaptive* because it readjusts the size of the memory regions for which it allocates counters as it gathers performance data.

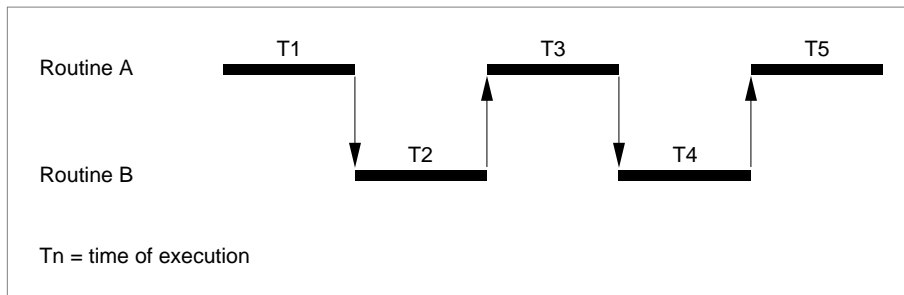
After you have run your application for some time and the ASP has collected its data, a statistical picture begins to form that you can then examine to determine which parts of your code are taking the longest or executing most often. To view this picture, you need to generate a report. At this stage, the ASP matches the address information furnished by the bucket hits it has collected with a symbolic information file for the code to be measured. It can then display the following information for each routine in your application:

The Adaptive Sampling Profiler

- address of the routine
- total time the routine has executed
- number of hits recorded for that routine
- percentage of time consumed by that routine

All time measurements given by the ASP are flat-time measurements. **Flat time** is the amount of time spent executing a routine. For example, given the routines A and B, where routine A calls routine B, as shown in Figure 4-1, the flat time for routine A is equal to $T1 + T3 + T5$.

Figure 4-1 Flat-time measurement



Because the ASP measures performance over the entire memory range, it can furnish rough statistics about the time spent executing Toolbox routines and routines in other fragments (for example, the Mixed Mode Manager, InterfaceLib, and shared libraries).

IMPORTANT

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 perhaps to resonance effects. See "Evaluating Performance Data" on page 4-15 for additional information. ▲

The ASP allows you to edit the data it has collected before you view the performance report. For example, you can have the ASP filter out information for regions of memory that are hit too seldom or too often (system calls).

The Adaptive Sampling Profiler measures systemwide performance. This means that you can measure code belonging to the current application as well as other code that might be running at the same time.

You can measure the performance of code residing in shared libraries or in a stand-alone code resource by loading the appropriate symbolic information files in the debugger. Then you can create an application whose only function is to call routines in the shared library or to load and execute the routines in the stand-alone code resource. Because the ASP records hits in the entire memory range, you will be able to collect and view performance data for nonapplication code.

Note

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

Using the Adaptive Sampling Profiler

The ASP is an integral part of the debugger. Use the debugger host to enable performance measurement and to generate a performance report. Before you can use the ASP, you must

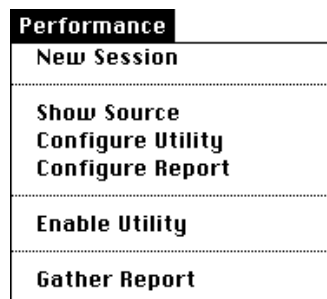
- launch the debugger host
- launch the application to be measured
- make sure that the symbolic information file or files for the code you are measuring reside where they are available to the debugger.

Before you can start a sampling session, the application must initially be stopped. You can stop your application using the “break-on-launch” method: press the Control key as you double-click the application. You can also stop the application after launch by inserting a call to the `DebugStr` routine as the first statement in your `main` function. A third method is to stop at a breakpoint.

Some of the sample applications included in the software developer kits include a menu item `Debug` that stops the application by calling `DebugStr`.

When the application is stopped, you can start an ASP sampling session. You control the ASP by choosing items from the `Performance` menu. Figure 4-2 shows the `Performance` menu.

Figure 4-2 The `Performance` menu



The following sections describe how you use the ASP to collect performance data and to generate a report.

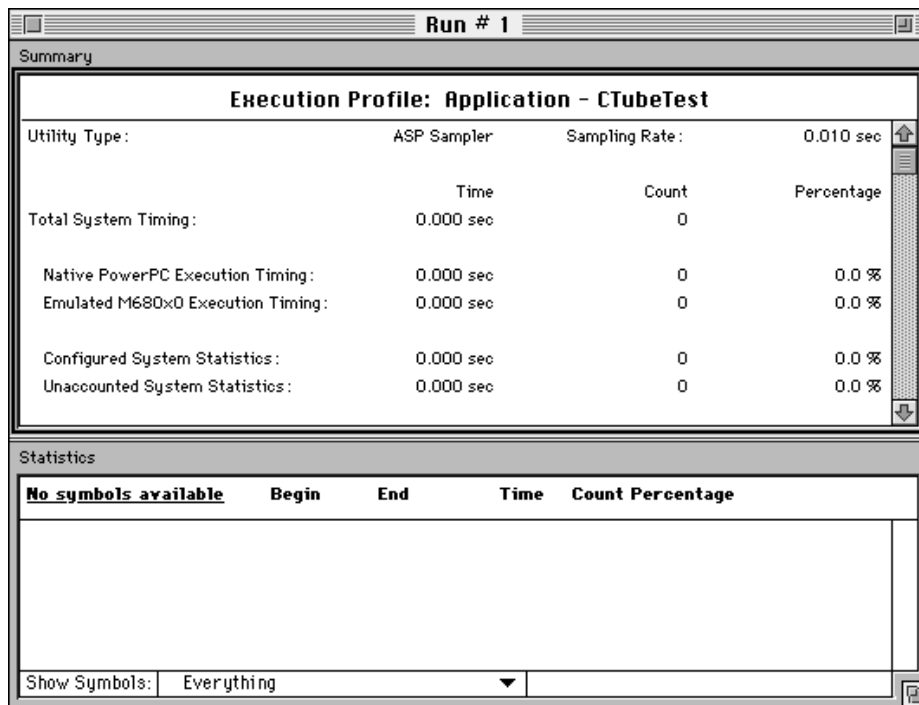
Enabling Performance Measurement

Before you can have the ASP collect performance data, you need to create a performance document and to specify a sampling rate. A **performance document window** is a window in which the ASP displays the performance data for a single performance session. The following sections explain how you do this.

Starting a Profiling Session

To create a performance document, choose New Session from the Performance menu. A window without any data in it, like the one shown in Figure 4-3, is displayed on the screen. The window's title identifies the performance session. In Figure 4-3 it is Run # 1. If you were to end this session and start another for the same application, the ASP would name the window for the second session Run #2. The automatic numbering lets you distinguish sampling sessions for the same application.

Figure 4-3 A blank performance document window



The name of the application to be measured appears just below the window title. The name of the application is the same as the name of the fragment that contains the main function for the current process.

The performance document window contains two scrollable views: the **summary view** and the **statistics view**. When you first open the performance document, before you have collected performance data and generated a report, these views do not have any data.

The Adaptive Sampling Profiler

- The summary view displays information about the total time the application has executed, the total number of sample hits, the sampling rate used for this invocation, and the number of hits for routines in shared libraries.
- The statistics view provides more detailed profiling information for each fragment sampled and, in the case of fragments for which symbolic information files exist, for each routine in the fragment.

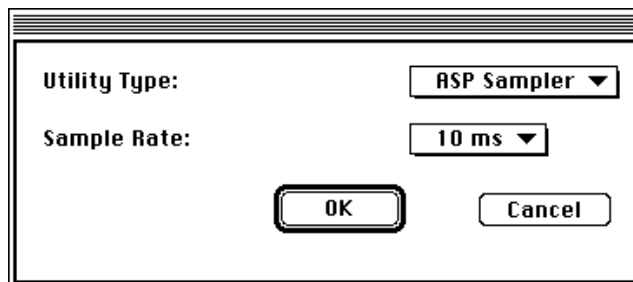
You can change the relative size of these two views by placing the cursor directly over the split lines and dragging the resize icon up or down to enlarge the desired view.

For more information about the summary and statistics views, see the sections “The Summary View” and “The Statistics View” on page 4-11.

Specifying a Sampling Rate

The sampling rate value for the current session is shown in the upper-right corner of the performance document window. This value is set to 10 milliseconds (ms) by default. If you want to specify a different sampling rate, choose Configure Utility from the Performance menu. A dialog box like the one shown in Figure 4-4 is displayed.

Figure 4-4 The utility configuration dialog box



A sampling rate of 10 ms means that every 10 ms the ASP will issue an interrupt and record the current PC value. To change the sampling rate, choose another value from the pop-up menu.

Take care in selecting a sampling rate. A time interval that is too large can result in a partial view of your program’s performance. A time interval that is too small can result in excessive interrupt processing that might perturb the execution of the system. There is no hard and fast formula for choosing a sampling rate. Instead, you should try multiple sessions with different sampling rates to arrive at the best overall picture of your program’s performance.

Unless you have a specific reason to change the sampling rate value, it is recommended that you begin by working with the default setting. For additional information about sampling rate values, see the section “Evaluating Performance Data” on page 4-15.

Once you have chosen a sampling rate and resumed execution, it is not possible to reconfigure the sampling rate.

Collecting Performance Data

After you have opened the performance document and selected the sampling rate, you are ready to have the ASP collect performance data. To do this, follow these steps:

1. **Choose Enable ASP Sampler from the Performance menu to begin measuring performance.**
2. **Choose Step or Run from the Control menu or the control palette to resume execution of the application.**

The ASP automatically collects data as you run your application. It's up to you to test specific features of your application or to test everything your application can do.

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

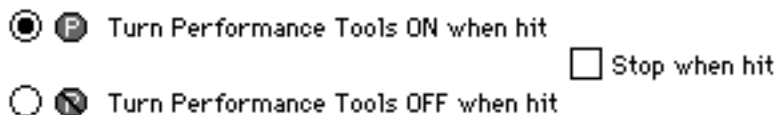
Measuring Selected Routines

You can restrict measurement to specific routines in your program by placing performance breakpoints around the routines you want to measure. To do this, follow these steps:

1. **Activate the browser window for the code you want to measure.**
2. **Place the cursor in the breakpoint column, next to the statement or instruction where you want to start sampling.**
3. **Press the Option key and click.**

The debugger displays a breakpoint dialog box. Part of this dialog box, shown in Figure 4-5, allows you to specify the type of breakpoint you want to set.

Figure 4-5 Selecting a performance breakpoint



4. **Click the Performance Breakpoint radio button and then click the Turn Performance Tools ON radio button.**
This places the performance-on icon in the breakpoint column.
5. **Find the statement or instruction where you want to stop sampling.**
6. **Option-click in the breakpoint column next to the statement or instruction. When the breakpoint dialog box is displayed, click the Performance Breakpoint radio button and then click the Turn Performance Tools OFF radio button.**
This places the performance-off icon in the breakpoint column.

The Adaptive Sampling Profiler

7. Repeat steps 2 through 6 as often as you need to insert performance breakpoints throughout your source file.

Take care when configuring performance measurement zones. There is some overhead associated with the communication of performance breaks, and time-critical paths (such as processing events in the event loop) could be significantly perturbed by placing such zones in them.

8. Create a performance document and select a sampling rate.**9. Run your program.**

The sampling data collected will be limited to the zones you have configured for performance measurement.

You can also enable and disable performance measurement manually by placing breakpoints in your program and selecting Enable Utility or Disable Utility from the Performance menu to enable and disable performance measurement when a breakpoint is reached. When the program is stopped, you can step through the code; if you enable performance measurement, the ASP can collect data in either state.

When you have finished, choose Gather Report from the Performance menu to generate a report for the data collected so far.

Generating a Performance Report

To generate a performance report for the data collected so far, stop the program on the target machine and return to the host machine. Choose Gather Report from the Performance menu. Once you do this, the ASP collects and sorts information for every routine in every fragment for which you have loaded a symbolic information file into the debugger. Once finished, the ASP updates the performance document window. Figure 4-6 shows a performance document window with data from a report.

How the ASP Presents Collected Data

While the ASP registers hits, it associates these hits with distinct memory ranges or buckets. All it knows about these ranges is their beginning and ending address. When the ASP generates the performance report, it tries to identify these memory ranges with specific fragments associated with your application and with routines within these fragments. If the ASP cannot associate a memory range with your application or with one of the fragments linked to your application, it accumulates all such hits in the category System/680x0 Emulated Symbols shown in the summary view.

How much information the ASP can display about hits collected inside or outside of a fragment depends on the information it can access about symbols associated with an address range:

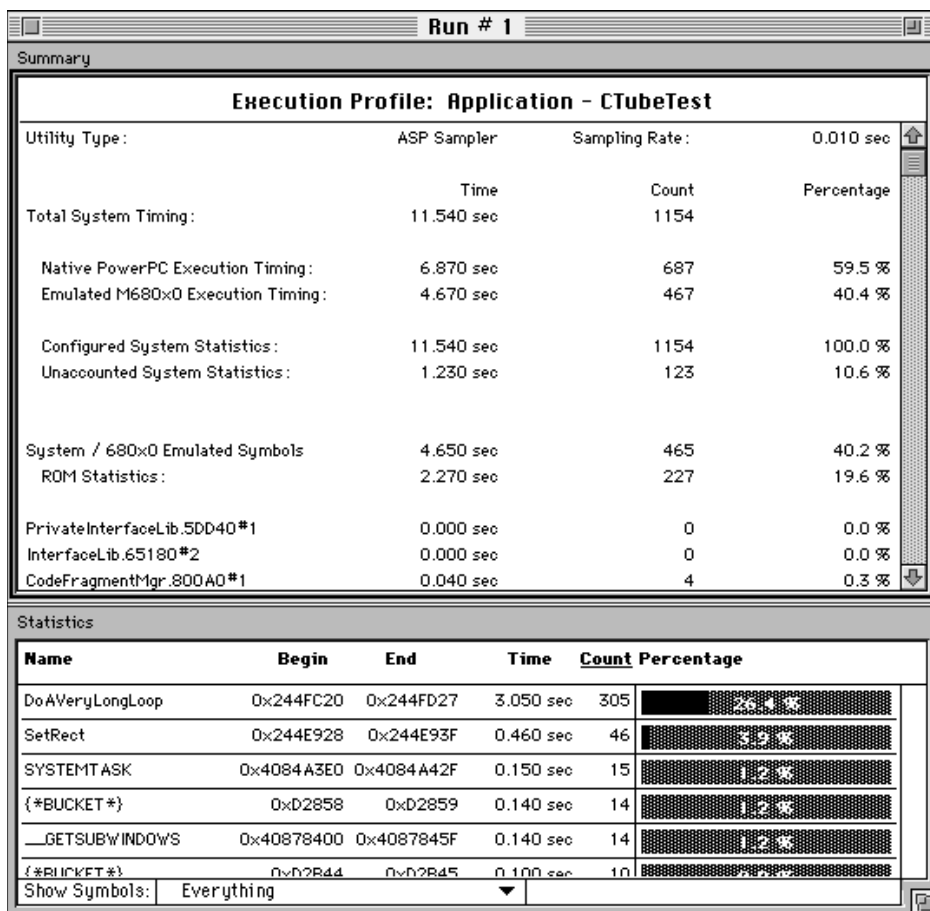
- If the hit occurs within an address range associated with one of the fragments linked to your application and there's a symbolic information file for that fragment, the ASP attributes the hit to one of the fragment's routines. If no symbolic information file exists for the fragment, the ASP looks for a MacsBug symbol whose address falls within the bucket containing the hit. If it finds such a symbol, it attributes the hit to it.

The Adaptive Sampling Profiler

If it does not, it attributes the hit to **bucket** and specifies the beginning and ending address of the bucket in the statistics view.

- If the hit occurs outside of an address range associated with one of the fragments linked to your application, the ASP checks to see whether the address lies in the ROM address range. If it does, it accumulates such hits in ROM Statistics (shown in the summary view). If it can identify that address with a system symbol, it attributes that hit to the system routine in the statistics view. If the address does not fall within the ROM range, the ASP tries to find a MacsBug symbol for that address, and if it finds such a symbol, it attributes the hit to it in the statistics view. If it cannot find a MacsBug symbol, it attributes the hit to **bucket** and displays the beginning and ending address of the bucket in the statistics view.

Figure 4-6 A performance document window after a report has been gathered

**Note**

The numbers that follow the fragment names are unique identifiers used by the debugger nub to distinguish between fragments having the same name. ♦

The Adaptive Sampling Profiler

The Summary View

The top line of the summary view identifies the utility that produced the current report and specifies how that utility was configured. In this case, the utility is the ASP Sampler, configured to sample the PC every 10 ms.

The next grouping of information shown in the summary view displays data in five categories: Total System Timing, Native versus Emulated Execution Timing, Configured System Statistics versus Unaccounted Statistics, System/680x0 Emulated Symbols, and ROM Statistics.

- Total System Timing represents the total number of hits recorded and the total time for those hits. The lines indented under Total System Timing represent different types of breakdowns for the total hits.
- Native PowerPC Processor Execution Timing and Emulated M680x0 Execution Timing present statistics that further break down the Total System Timing data according to the native or emulated PCs sampled.
- Configured System Statistics represents the total number of hits that the ASP was able to map to a specific fragment. It also represents the difference between Total System Timing and Unaccounted System Statistics, which represents the total number of hits accumulated either as a result of ASP registration errors or as a result of filtering criteria.

ASP registration errors occur when the address range of a hit overlaps the boundary of an executable fragment. In such cases, the ASP cannot definitively map a hit to a specific fragment and assigns such hits to Unaccounted System Statistics.

If you use the report configuration dialog box (shown in Figure 4-9 on page 4-14) to exclude information from the final report, hits accumulated for the excluded regions are included in Unaccounted System Statistics.

- System/680x0 Emulated Symbols represents the number of hits collected for code executing outside of any fragment associated with the application being measured. Most applications spend a good deal of time in this region, especially applications that make heavy use of the user interface. The address ranges associated with this section include the ROM.
- ROM Statistics represents the total number of hits that were associated with a ROM address range. If there are system symbols or MacsBug symbols associated with a ROM address range, more detailed information for these routines is shown in the statistics view.

Following these summaries is a summary breakdown of statistics per fragment. Each line displays summary information for one fragment. These are listed in ascending order by the beginning address of each fragment.

The Statistics View

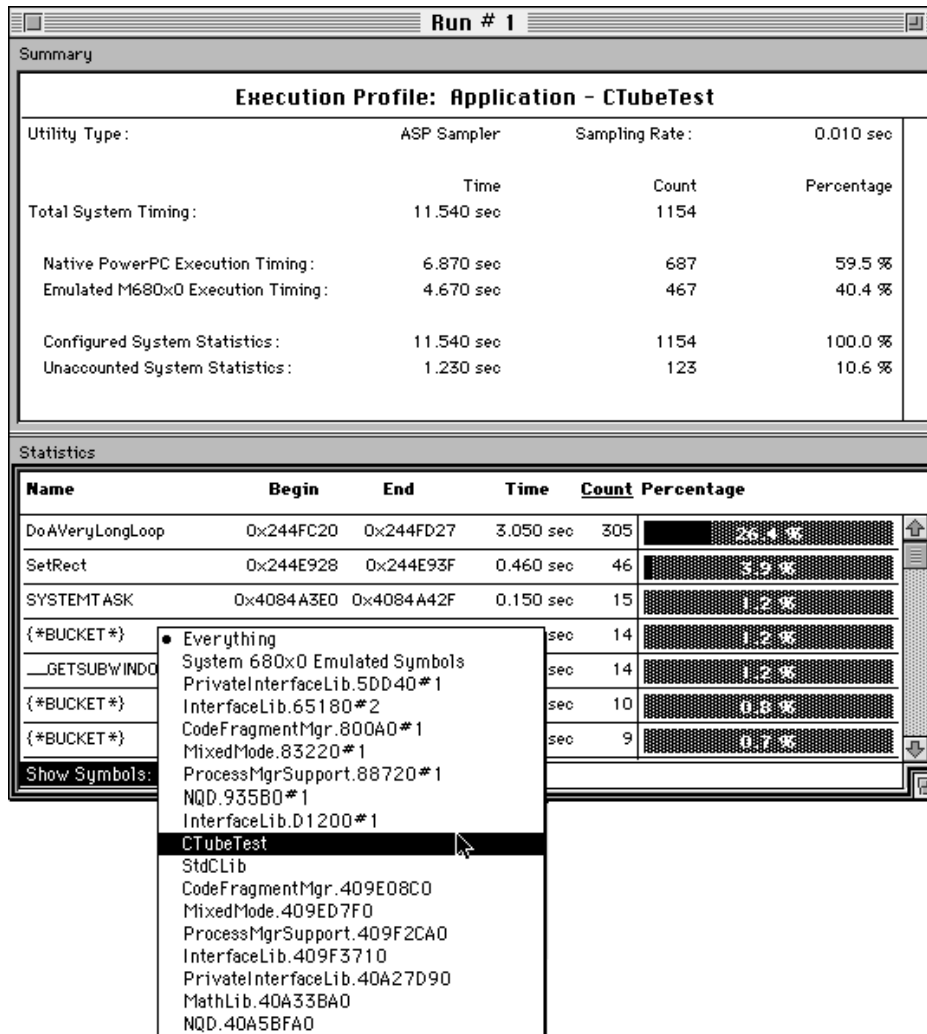
The statistics view displays information for all fragments called by your application and for all routines in those fragments if a symbolic information file for the fragment has been loaded into the debugger. This default setting is signaled by the label Everything in the Show Symbols pop-up menu in the lower-left corner of the performance document. Also by default, the statistics displayed for the modules and fragments in the statistics

The Adaptive Sampling Profiler

view are listed in descending order according to the number of hits registered for each fragment.

To display data for a specific fragment, choose the name of the fragment from the Show Symbols pop-up menu. Figure 4-7 shows the fragment CTubeTest being selected.

Figure 4-7 Displaying data for a fragment



To change the sort order for the data displayed, click one of the column headings Name, Begin, End, Time, Count, or Percentage. The heading you select is underlined.

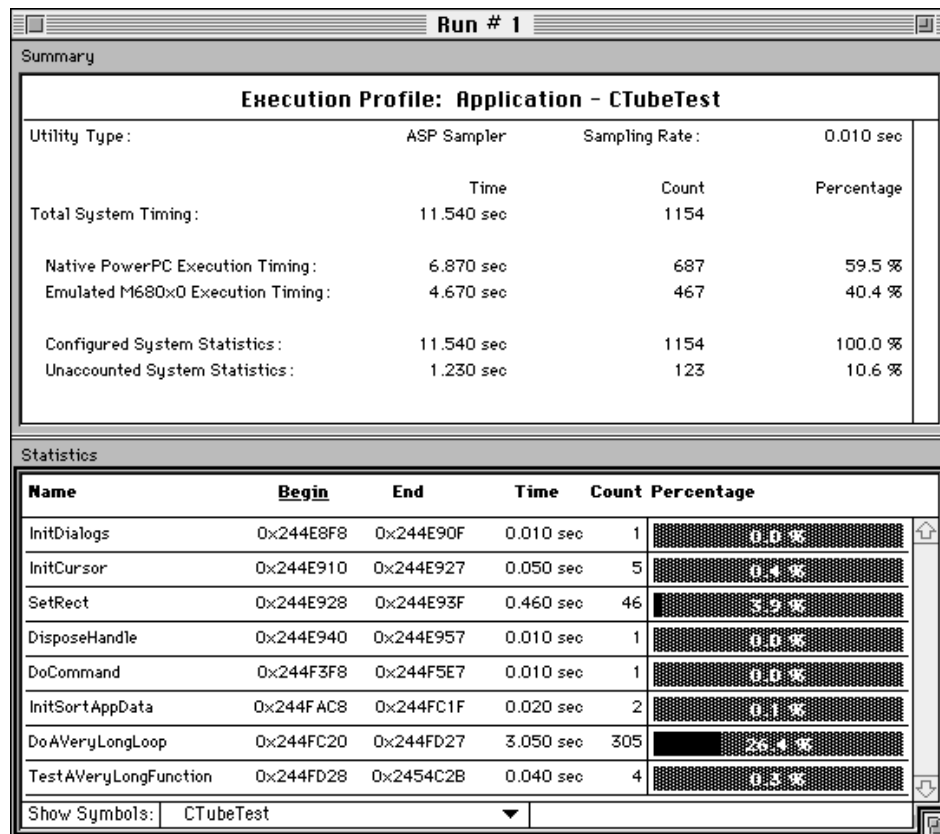
- Name sorts the data in alphabetical order by module and fragment name.
- Begin sorts the data in ascending order by address: statistics for the module or fragment at the lowest address are shown first.

The Adaptive Sampling Profiler

- Time, Count, and Percentage sort data in descending order, with the most frequently used shown first.

Figure 4-8 shows data sorted by beginning address in the statistics view.

Figure 4-8 Data sorted by beginning address



In certain cases, the ASP is not able to attribute a name to a memory region where a hit was registered. For additional information, see “How the ASP Presents Collected Data” on page 4-9.

Editing the Performance Report

To have the ASP filter out information based on performance data, choose the Configure Report menu item from the Performance menu. The ASP displays a report configuration dialog box like the one shown in Figure 4-9.

Figure 4-9 The report configuration dialog box

☒ **Percentage :**
Min : % **Max :** %

☒ **Count :**
Min : **Max :**

☒ **Time :**
Min : ms **Max :** ms

This dialog box consists of statistics filters. You supply minimum and maximum values for one or more of these in order to filter out information based on performance criteria.

1. Click one of the checkboxes labeled **Percentage**, **Count**, or **Time** on the left side of the dialog box.

Leaving a checkbox blank means that this filter is inactive.

2. Specify a minimum and maximum value in the text fields labeled **Min** and **Max**.

For example, if you select **Percentage** and specify 1% as a minimum value and 80% as a maximum value, the ASP will not display information for routines that have received less than 1% or more than 80% of the hits gathered. Leaving the **Min** or **Max** text field blank means that the field has no bounds.

It is possible to 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, the ASP will filter out information according to whichever of these values is the final minimum.

Saving and Printing a Performance Document

You can save a performance document by choosing **Save Performance** from the **File** menu. The document is saved as a tab-delimited MPW text file. You can view the file using MPW or Microsoft Excel. Currently, it is not possible to reload and view the file using the Macintosh Debugger.

To print the statistics view of a performance document

1. Make the performance document the active window.
2. 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 this data. Because some measurement results might not make sense to you at first, this section offers some suggestions on interpreting your performance data.

- The ASP shows a hit for a function that I know could not possibly have been called in this session. What is wrong?

Nothing is wrong. While tracking and splitting buckets, the ASP sometimes has to arbitrarily distribute a very small number of bucket hits over a range of memory. As a result, the ASP might record a hit for a routine that was not called. In using the ASP, you really want to focus on those parts of your code that are used most often. For those parts, the number of false hits relative to the number of real hits is statistically insignificant. For a more detailed look at how the ASP gathers information, see “How the Adaptive Sampling Profiler Gathers Data” on page 4-15.

- I know that I am calling a function many times, but the ASP only shows hits for one source line in that function. Is the ASP broken?

No, it isn't. You probably have a resonance effect, which can occur when an application's source 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 source code line in that loop. The situation is like viewing a spinning wheel with a strobe light as the light source. If the strobe light pulses at the same rate or an even multiple of the spinning wheel's period, the wheel will appear to be motionless. In your case, the ASP is the strobe light and the loop in your code is the spinning wheel.

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

- How can I tell when I should stop gathering data?

Given that performance measurement is an iterative process, you can tell you have sufficient information when the statistics you gather begin to converge. For example, if the statistics for some routine show a percentage of 2% during one performance session, 4% for another, and 6% for another, you do not have sufficient convergence. When the percentage figures are more equal (say 5%, 6%, and 6%), you can stop measuring.

How the Adaptive Sampling Profiler Gathers Data

This section describes how the ASP registers hits and how it matches hits with distinct memory regions. You do not have to read this section to use the ASP. However, understanding how the ASP performs its measurements can help give you a better understanding of the meaning of some of its data.

The ASP registers hits that occur anywhere in memory. Because the ASP is a systemwide performance measurement tool, it allows you to measure the extent to which your application interacts with code throughout the system and also allows you to measure the performance of code other than applications. However, taking measurements across

The Adaptive Sampling Profiler

the entire range of memory means that the ASP must use a special strategy for associating counters with memory regions. This *adaptive* strategy is a way of dynamically subdividing ranges of executable code addresses during the execution of a program according to the frequency with which particular addresses in those ranges are executed.

For example, suppose your application resides in the range 0x4000 through 0x5000 and there are no other active applications running. In this case, the largest density of hits will occur in your application's memory range. Traditional sampling tools would divide the entire sampling range into a large number of buckets. Each bucket would be associated with the same size of memory. This results in having to allocate a lot of memory (for the buckets) while at the same time achieving a fairly gross level of measurement granularity for the regions of high execution density. In contrast, adaptive sampling registration creates only a small number of buckets (associated with large memory-range areas) for low-density areas of execution. Conversely, it creates a larger number of buckets (associated with small memory-range areas) for high-density areas of execution. In our example, the ASP might create 2 buckets, each containing one hit for the addressable range of 0x000 through 0x3FFF, 16 buckets spanning the range from 0x4000 through 0x4FFF containing 200 hits, and 5 buckets spanning the range from 0x5000 through the maximum address containing 1 or 0 hits.

Using adaptive sampling registration, the boundaries of the address regions can change over the course of the application's execution as the number of buckets associated with high-density execution areas is increased. The next three sections explain in greater detail how this is done.

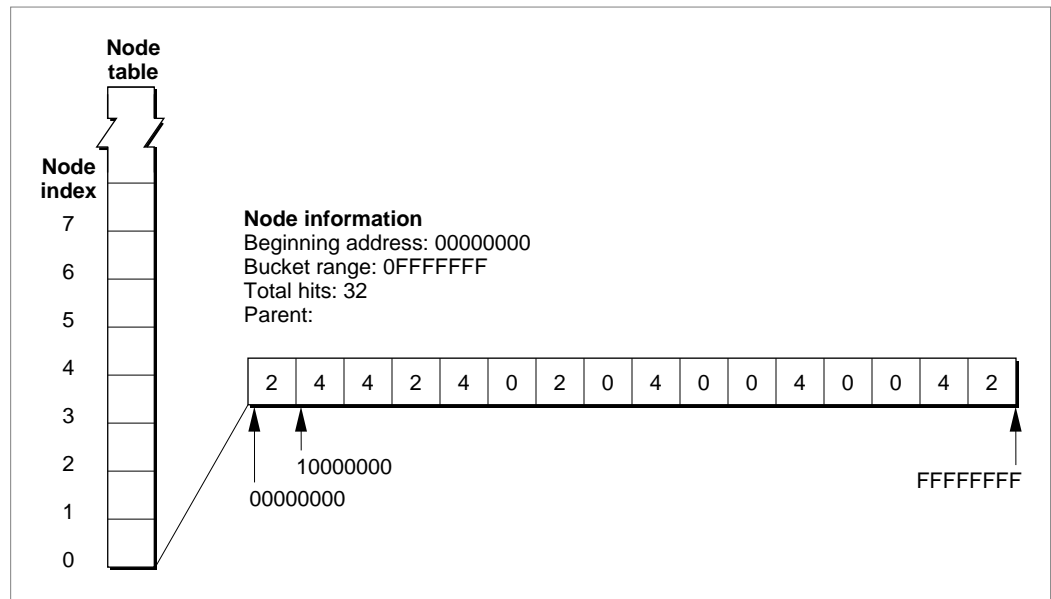
Allocating Nodes

The ASP uses an array called a **node table** to track the location of PC samples and to identify the most exercised parts of your code. Each node in the node table contains information that identifies that node: this information includes the address range associated with that node and the number of hits occurring in that range.

When sampling begins, hits are registered in the first node (node index 0) of the node table. This node is divided into 16 buckets that cover the entire address space (0000 0000 to FFFF FFFF). Each bucket has a range of 0FFF FFFF.

When the ASP receives an interrupt with a PC address, it registers the hit in the first node's total hits field. The ASP then determines the correct bucket range for the PC value registered and increments the bucket count for that range.

Figure 4-10 shows an expanded picture of the first node in the node table after 32 samples have been taken. As you can see, a count for each memory region has been registered in the bucket corresponding to that region.

Figure 4-10 Recording bucket hits in a node

Splitting Buckets

After the ASP has incremented a bucket, it determines whether the bucket should be split by comparing the bucket count to a dynamically calculated threshold number. The following formula is used for calculating this number:

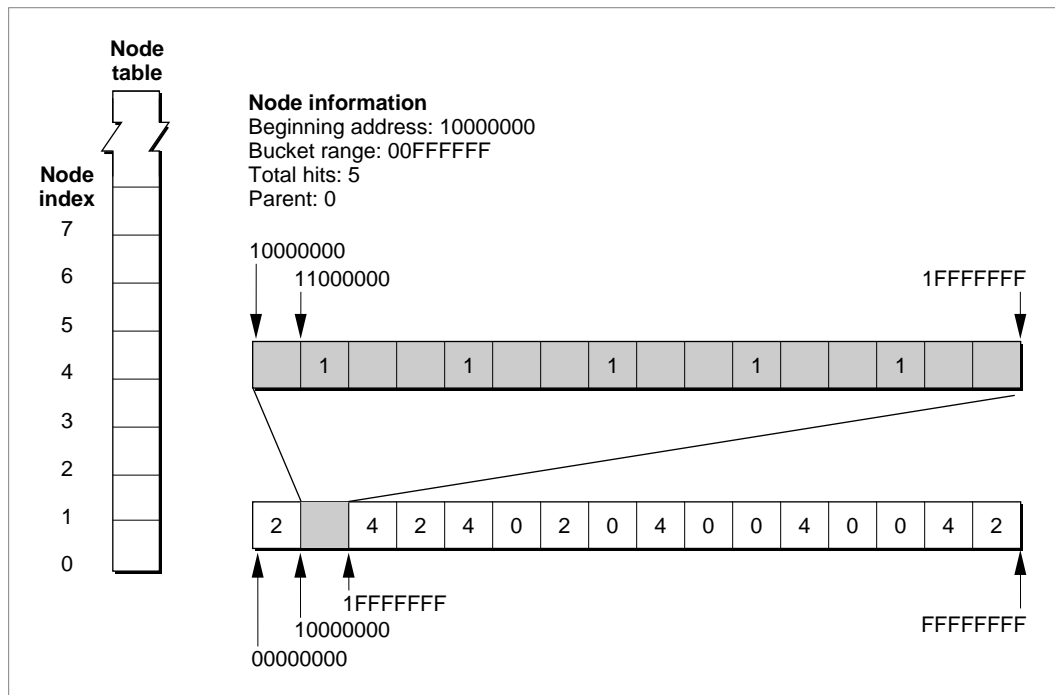
$$T = \max(5, (S/128))$$

Where T specifies the threshold value and S specifies the total number of samples taken.

As you can see, for a small number of samples the threshold value is 5. That is, a bucket is split when its count reaches 5. As the number of samples taken increases, the threshold also increases. Thus, when the number of samples increases to 768, the threshold value is 6; when the number of samples is 896, the threshold value is 7, and so on.

For example, with reference to Figure 4-10, suppose that one more sample is taken and the counter in the second bucket from the left increases to 5. This is the defined threshold limit. Accordingly, the ASP splits the bucket. It acquires a new node from the node table and inserts information into the node that was split to link it to the parent node.

Figure 4-11 shows what the node table and two allocated nodes look like after the bucket is split.

Figure 4-11 Splitting a bucket

The node information in the new node records the beginning address for the memory range corresponding to the new node as well as a new bucket range for that node. Note that the buckets of the new node are smaller by one order of magnitude (a factor of 16) than the buckets of the parent node. Depending on the number of hits and the threshold value, the ASP continues to split buckets until it reaches the smallest memory range for which it can record hits: 1 byte per bucket for 16 buckets.

What has happened to the count of total hits for the bucket after it is split? In this case, 5 hits were registered. When the ASP splits a bucket, it no longer has the PC information for the hits in that bucket, so it evenly distributes them to buckets in the new node, as shown in the previous figure. As you can see, the process of splitting buckets might result in a hit being recorded for a memory range that has not in fact been hit. Because the sampling information in which you are interested is always for the parts of your code that are used most often, this even distribution should not present a problem.

The ASP tool uses a fixed-size node table to record hits. The dynamic allocation of nodes and the dynamic calculation of threshold values allows the ASP to use the greatest precision in registering hits for the parts of your code that get the most use. The two drawbacks to this method are that the granularity is quite large for memory regions where the least hits are recorded and that there is some information loss when buckets are split. Nevertheless, this method also ensures the finest granularity for the areas in which you are interested, while the information loss relative to the number of samples taken is negligible.

Freeing Nodes

The ASP allocates new nodes from the node table whenever it needs to split a bucket. If all entries in the node table have already been allocated and a bucket needs to be split, the ASP selects five of the least-frequently used nodes in the node table and frees these nodes from the table. The hits that had been accumulated in these buckets are reallocated to the parent node and additional hits, from a different memory range, can now be accumulated in these nodes.

Registration Errors

Registration errors do occur during adaptive sampling registration, but it is important to remember that these errors are statistically insignificant. Registration errors can occur when mapping functions, subdividing address ranges, and performing garbage collection (freeing nodes). These three types of error are called function-mapping errors, dynamic splitting errors, and dynamic reformation errors.

Function-mapping errors are inherent in all sampling tools. This type of error occurs when there are buckets that have address ranges that do not correspond to the functional boundaries of the code being measured. Buckets might have address ranges that overlap a function entrance or exit, totally envelop a function or functions, or get enveloped by a function. When addresses are attributed to functions, it is done statistically. This means that the number of hits credited to a function is equivalent to the percentage of code that function has in the memory range times the number of hits in the memory range.

Dynamic splitting errors can occur when the number of samples an address range has received exceeds a threshold and the address range is divided into subranges (bucket splitting). The hits accumulated in the address range no longer have PC information associated with them, because the bucket accumulates only the number of hits. When the bucket is split, the existing count is spread across all subdivisions, even though some of those subdivisions may not have received hits.

Dynamic reformation errors occur when the ASP has allocated all available buckets but requires more buckets to continue recording hits. It acquires the needed buckets through a garbage collection mechanism. It looks for the buckets containing the lowest number of hits, obtains a total of all counts in its subranges, and moves this total to the address area of the bucket's parent. This means that some specific information about where samples have been collected may be lost. Because the bucket's address range (and subranges) have a low density of hits, this error is insignificant.

ASP Overhead

In addition to the kinds of registration errors described in the previous section, you need to be aware of the normal overhead generated when measuring an application's performance with the ASP. There are two kinds of overhead incurred when using the ASP, total overhead and measured overhead.

The **total overhead** incurred when measuring performance using the ASP is the total difference in the elapsed times when running your application with the ASP and

The Adaptive Sampling Profiler

running your application without the ASP. Tests have determined that the total overhead associated with the ASP is about 3% of the total time.

The **measured overhead** incurred when measuring performance using the ASP is that portion of the total overhead contained in the sample timings themselves. Tests have determined that the measured overhead associated with the ASP is less than or equal to 2% of the total time.

Low-Level Debugging

Contents

Memory	5-3
Data Storage	5-3
Data Alignment	5-4
Locating a Fragment's Code and Data in Memory	5-4
Obtaining Heap Information	5-7
Registers	5-7
The Stack	5-9
680x0 Stack Frames and Calling Conventions	5-12
PowerPC Stack Frames and Calling Conventions	5-12
System-Level Debugging	5-14
Debugging Memory-Based Fragments	5-14
Mixed-Mode Debugging	5-15

Low-Level Debugging

This chapter supplements information presented in Chapter 3, “Using the Debugger.” It explains

- the conventions used to store multibyte quantities in memory
- the conventions used to align data
- how to locate a fragment’s code and data sections in memory
- the conventions regarding the use of the PowerPC registers
- the conventions governing the construction of stack frames on the PowerPC
- how to catch calls to system software routines
- how to debug memory-based fragments
- how to do mixed-mode debugging

In short, whereas Chapter 3 focused on making you familiar with the debugger interface, this chapter aims to add some depth to that picture by relating the information displayed by the debugger with the PowerPC run-time architecture. The information presented in this chapter can also help you debug programs for which you do not have symbolic information. This chapter does not discuss RISC chip architecture or the RISC instruction set and addressing modes. For information on these topics, see *Assembler for Macintosh With PowerPC* or Motorola’s *PowerPC 601 RISC Microprocessor User’s Manual*. For information on the PowerPC run-time architecture, see *Inside Macintosh: PowerPC System Software*.

Memory

A computer’s address space consists of the total amount of memory the microprocessor is capable of addressing. Every byte of memory in the address space has a unique address. On the Power Macintosh computer, addressable memory ranges from \$0000 0000 to \$FFFF FFFF, allowing up to 4 gigabytes (GB) of storage. You can use the debugger’s memory display to examine the contents of any valid address in that range.

Data Storage

If the data stored in memory were always 1 byte long, reading the memory display would be quite easy. In reality, most data is almost always larger than a byte, which raises the problem of how successive bytes are to be arranged in memory and thus of how to interpret the values shown in the memory display window. The 680x0-based Macintosh and the Power Macintosh use the same convention in storing multibyte values: the most significant byte is stored at the lowest address. For example, a long word (4 bytes) would need to be stored at four successive addresses.

Data Alignment

The PowerPC processor's speed is partly due to its use of aligned loads and stores. This means that the processor can read from and write to memory much more quickly if the compiler stores multibyte quantities in the following ways:

- Half words (16 bits) begin on an address that is a multiple of 2.
- Words (32 bits) begin on an address that is a multiple of 4.
- Double words (64 bits) begin on an address that is a multiple of 8.
- Quad words (128 bits) begin on an address that is a multiple of 16.

If you port a 680x0 program to a Power Macintosh, you might find that your data structures have "grown" due to padding inserted by the compiler to enable aligned loads and stores. For example, the following structure would take up 8 bytes on a 680x0-based Macintosh, but 10 bytes on a Power Macintosh.

```
struct misaligned {
    short version;          /*2 bytes of padding after this field*/
    long address;
    short count;
};
```

A Power Macintosh can perform misaligned loads and stores but may have to give up some speed to do so. If you need to exchange data structures on disk between 680x0 and PowerPC versions of your program, if you need to pass structures to emulated code, or if you pass custom structures to the Toolbox, you can do one of the following:

- Use the appropriate compiler directive that preserves 680x0 alignment for that structure.
- Emulate PowerPC alignment for your aggregate types using zero-length bit fields and explicit padding. You can also organize your structures in such a way that there is no difference between the 680x0 and PowerPC alignment. Usually this means placing the largest (integer-sized) fields first, followed by shorts, followed by chars.

Note

Access to misaligned data on the PowerPC is not necessarily slower than access to aligned data. Access to misaligned data is slower if it spans certain boundaries. For example, if the misaligned data spans a 64-bit (double-word) boundary, it takes two data transfers to access the data. If the misaligned data spans a cache line, page, or segment boundary, accessing this data might raise an alignment exception, which could take a substantial amount of time. However, these latter cases are rare. ♦

Locating a Fragment's Code and Data in Memory

The run-time architecture of the Power Macintosh computer differs markedly from the run-time architecture of the 680x0 Macintosh computer. One result of this difference is that your application's code and data are organized differently in memory on the Power

Low-Level Debugging

Macintosh. This section summarizes some of these differences and explains how you can locate the different parts of a fragment in memory.

On a 680x0 Macintosh, your application's code and data reside in the application's heap in RAM. The code, which is segmented into code resources, is loaded into memory as it is needed and unloaded when it is no longer required. This scheme came into being because early Macintosh computers had only a small amount of memory.

A **fragment**—any block of executable PowerPC code and its associated data—is divided in two sections: a data section, which contains the fragment's static data as well as the TOC, and a code section, which contains the fragment's code. The fragment, in turn, might reference data or routines (imports) stored in other fragments; it accesses these values through entries in its TOC that are resolved at run time by the Code Fragment Manager. The Code Fragment Manager loads an application's data section into the application's heap; it loads the data section of a shared library containing only data that is globally shared into the system heap. A fragment's code section is marked read-only and placed in one of four locations depending on whether virtual memory is on and on the type of the fragment.

- If virtual memory is on, your application's code is file mapped to a range of logical addresses above the address pointed to by `bufPtr`.
- If virtual memory is off (or virtual memory mapping fails) and the fragment is an application, the Code Fragment Manager loads the code section into the application's heap.
- If virtual memory is off (or virtual memory mapping fails) and the fragment is a shared library or code extension, the Code Fragment Manager attempts to load the code section into the Process Manager's temporary memory. If temporary memory allocation fails, the code section is loaded into the system heap.

It's important to remember that the data and code sections might be discontinuous. Thus, for example, there is nothing wrong if the address of an instruction lies at some distance from your application's heap.

In debugging, it's often useful to determine where your application's code and data reside in order to identify possible garbage values, uninitialized pointer variables, and so on. To determine where your application's code resides, you can use the debugger's Show Fragment Info command in the Views menu. The Fragment Info window (shown in Figure 5-1) displays a list of all the fragments on the target machine.

Figure 5-1 Fragment Info window

Fragment Info					
Process Name	Section Name	Address	Size	Dbg?	Type
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$00682750	\$000ADB1C	Yes	Code
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$007487E0	\$000287FA	Yes	Data
DemoDialogsPPCMrC	ObjectSupportLib	\$00886A40	\$00003E74	Yes	Code
DemoDialogsPPCMrC	ObjectSupportLib	\$00145B70	\$00000278	Yes	Data
DemoDialogsPPCMrC	StdCLib	\$00D76170	\$0000EBC4	Yes	Code
DemoDialogsPPCMrC	StdCLib	\$007460B0	\$00002710	Yes	Data
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$40A03B40	\$00022E2C	Yes	Code
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$000153C0	\$00007384	Yes	Data
DemoDialogsPPCMrC	MathLib.40A33BA0	\$40A34AF0	\$000120D8	Yes	Code
DemoDialogsPPCMrC	MathLib.40A33BA0	\$00743910	\$00002778	Yes	Data
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$409EDEA0	\$00004438	Yes	Code
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$00011D50	\$00000EAB	Yes	Data
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$40A2E900	\$00004C50	Yes	Code
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$0000F320	\$00002A08	Yes	Data
DemoDialogsPPCMrC	MixedMode.70B60#1	\$00071230	\$000044A8	Yes	Code
DemoDialogsPPCMrC	MixedMode.70B60#1	\$000E1260	\$00000EB0	Yes	Data
DemoDialogsPPCMrC	ProcessMgrSupport.606B0#1	\$00060920	\$00000AA3	Yes	Code
DemoDialogsPPCMrC	ProcessMgrSupport.606B0#1	\$000E2130	\$00000158	Yes	Data

Refresh Show Exports

The Fragment Info window contains a lot of information that can be of use in debugging.

- The first column, Process Name, lists the name of the process associated with this fragment.
- The second column, Section Name, lists the section name for this fragment.
- The third column, Address, gives the physical address of the fragment.
- The fourth column, Size, gives the size of the fragment.
- The fifth column, Dbg?, says “yes” if the fragment is currently targeted for debugging; otherwise, it says “no.”
- The sixth column, Type, gives the fragment type: data or code.

Use the Show Exports button on the Fragment Info window (double-click on any row) to list all the exports for the selected fragment, as shown in Figure 5-2.

Figure 5-2 Show Exports window

StdCLib		
Symbol Name	Address	Symbol Type
abort	\$00746CA4	Code Vector
abs	\$00746518	Code Vector
access	\$00746B54	Code Vector
asctime	\$00746524	Code Vector
atexit	\$00746FE0	Code Vector
atof	\$00746530	Code Vector
atoi	\$0074653C	Code Vector
atol	\$00746548	Code Vector
binhex	\$00746554	Code Vector
bsearch	\$00746560	Code Vector
calloc	\$0074656C	Code Vector
clearerr	\$00746998	Code Vector
clock	\$00746578	Code Vector
close	\$00746B60	Code Vector
ConvertTheString	\$00746830	Code Vector

Low-Level Debugging

Note that the beginning address of fragments that are referenced by the application and by the system is exactly the same because the code section of each shared library is loaded at one address and all importing fragments reference that copy.

A fragment's data section is located within reasonable distance of the address stored in its Table of Contents Register (RTOC). This is because a fragment's TOC is stored in its data section and the RTOC points to the TOC.

Obtaining Heap Information

Although the debugger does not provide heap information, you can obtain this information for your fragment's heap by dropping into MacsBug on the target machine and executing MacsBug's heap commands. For additional information, see the *MacsBug Reference and Debugging Guide*.

Registers

In addition to addressable memory that is external to the microprocessor, the PowerPC processor can also access memory in its own registers. These include general-purpose registers R0 through R31, floating-point registers FPR0 through FPR31, and an additional number of special-purpose registers. You can display or change the contents of a register by using the registers and FPU registers displays. A register is either volatile or nonvolatile; a nonvolatile register is guaranteed to have the same value upon return as before a call.

Table 5-1 summarizes the conventions governing the use of general-purpose registers across call boundaries.

Table 5-1 The general-purpose registers

GPR	Volatile/ nonvolatile	Use
R0	Volatile	In glue and prologs while setting up or tearing down stack frames. While a function is executing, it is used as a scratch register. When used as a base register for addressing memory, it designates an absolute address. For example 4 (r0) specifies the address 0000 0004.
R1	Nonvolatile	Stack pointer (SP); points to top of stack.
R2	Nonvolatile	TOC pointer (RTOC); points to the TOC of the currently executing routine.

Low-Level Debugging

Table 5-1 The general-purpose registers (continued)

GPR	Volatile/ nonvolatile	Use
R3–R10	Volatile	To pass non-floating-point parameter values or pointers to structures. Register R3 is used for non-floating-point results.
R11, R12	Volatile	Scratch registers, used by system calls in glue and prologs for RTOC switch and pointer-based calls.
R13–R31	Nonvolatile	Storage of local variables and compiler temporary variables.

Table 5-2 summarizes the conventions governing the use of floating-point registers.

Table 5-2 The floating-point registers

FPR	Volatile/ Nonvolatile	Use
FPR0	Volatile	Scratch register.
FPR1–FPR13	Volatile	To pass first 13 floating-point parameters. The registers FPR1 and FPR2 are used for floating-point results.
FPR14–FPR31	Nonvolatile	Storage of floating-point local variables.

The contents of the Condition Register are shown in the registers display. The Condition Register is divided into eight fields of 4-bit values. In general, the Condition Register is used to control conditional branches. The Branch Unit examines the contents of the Condition Register to determine the outcome of branches. Table 5-3 summarizes the conventions governing the use of the Condition Register fields.

Table 5-3 The Condition Register

CR field	Volatile/ nonvolatile	Use
0–1	Volatile	Scratch fields set by fixed-point and floating-point operations via the record bit (Rc).
2–4	Nonvolatile	Local storage.
5	Volatile	Local storage. (Avoid using this register.)
6–7	Volatile	Scratch fields.

The contents of the special-purpose registers—the program counter (PC), Link Register (LR), Count Register (CTR), and Fixed-Point Exception Register (XER)—are displayed in the registers display. Table 5-4 summarizes the use of these registers.

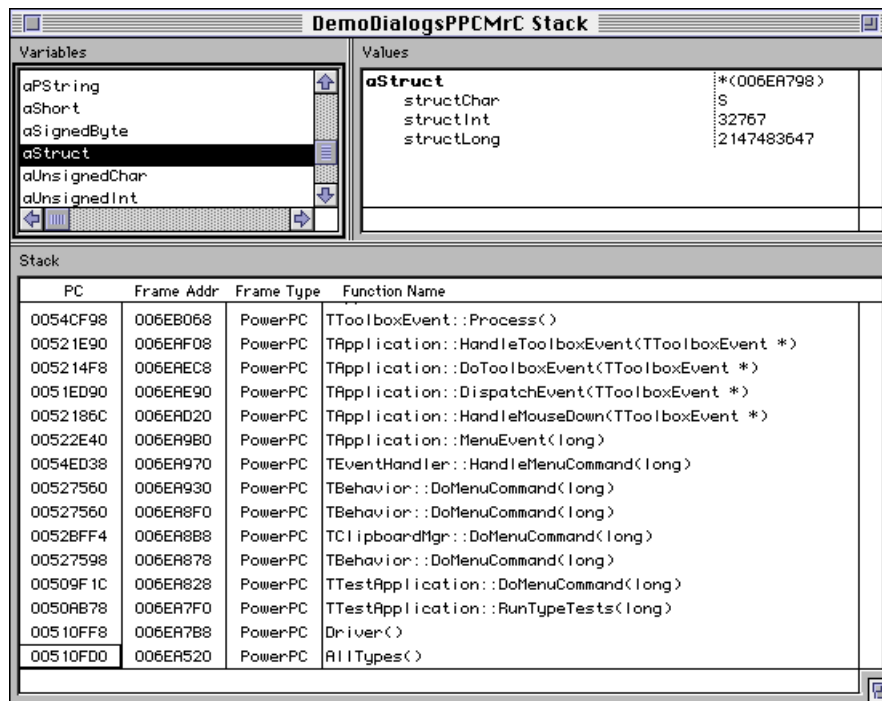
Table 5-4 The special-purpose registers

Register	Volatile/ Nonvolatile	Use
PC		Holds the address of the next instruction to execute.
LR	Volatile	Procedure call and return.
CTR	Volatile	Loops and computer branches.
XER	Volatile	Fixed-point status information.

One additional special-purpose register, the **Floating-Point Status and Control Register (FPSCR)**, is displayed in the floating-point registers window. The FPSCR handles floating-point exceptions and records the status of floating-point operations.

The Stack

The cooperative nature of the PowerPC run-time architecture is nowhere more visible than when looking at the stack, which accommodates both 680x0 and PowerPC frames while ensuring a seamless transition between the two through the use of a mixed-mode switch frame. You display stack information by choosing Show Stack Crawl from the Views menu. “The Stack” on page 3-23 provides summary information about the use of this display and the meaning of the information it contains; this section provides additional detail about the display as well as information that might be of use to you if you are looking at the stack without symbolic information. Figure 5-3 shows a sample stack display.

Figure 5-3 The stack display

The PowerPC stack is a grow-down stack, like the 680x0 stack. As a result, the most recently executed routine is at the “top of the stack,” that is, toward low memory. The information displayed for each frame has a different meaning depending on whether the frame is a 680x0 frame (designated by the label 68K in the Frame Type column) or a PowerPC frame. Table 5-5 describes the meaning of each column for 680x0 and PowerPC frames. PowerPC and 680x0 stack frames use a different format and different calling conventions. Understanding these differences can help if you need to examine stack frames for code that cannot be mapped to its source. The next two sections, “680x0 Stack Frames and Calling Conventions” and “PowerPC Stack Frames and Calling Conventions,” describe the special characteristics for each type of frame.

Table 5-5 Stack frame information

Frame type	PC	Frame address
680x0	<p>The address of the instruction following the JSR that calls the next function.</p> <p>If this 680x0 frame is the most recent frame, the PC value represents the current value of the PC.</p>	The value contained in the A6 register when this frame is active.
PowerPC	<p>The value of the program counter when this function calls the next function.</p> <p>If this PowerPC frame is the most recent frame, the PC value represents the current value of the PC.</p>	The value contained in the R1 register when this frame is active.

There are two additional types of frames allocated on the stack that are not visible in the stack display: switch frames and the Red Zone.

- When a 680x0 routine calls a PowerPC routine, or when a PowerPC routine calls a 680x0 routine, the Mixed Mode Manager creates a **switch frame** and inserts it between the 680x0 frame and the PowerPC frame on the stack. The format of the switch frame differs depending on whether the call is made by an emulated routine or by a PowerPC routine. Because the switch frame is partly used to pass parameters to the called routine in a way that the routine understands them, the size of the frame necessarily varies with the number of parameters it is passing. For detailed information about switch frames, see the Mixed Mode Manager chapter of *Inside Macintosh: PowerPC System Software*.
- The **Red Zone** is an area located just below the stack pointer, that may be used by a frameless leaf routine (a routine that does not call other routines and does not declare any local variables). Such a routine can use up to 224 bytes below the stack pointer to save nonvolatile registers.

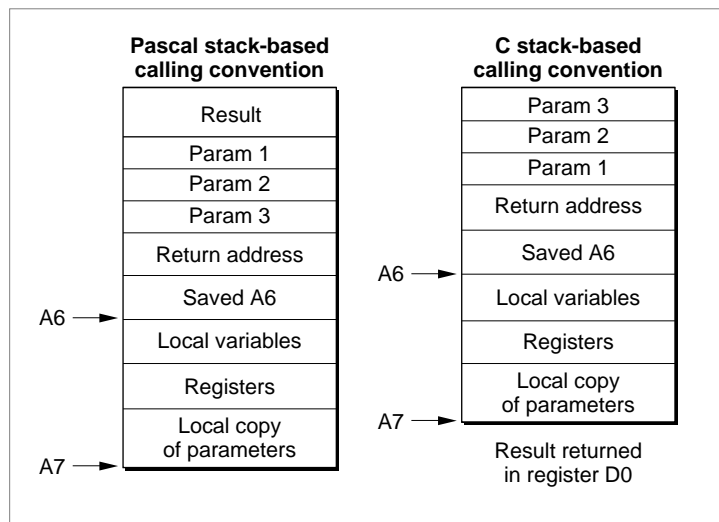
Of course, you can view both these frames in the memory display by dumping memory from the appropriate address.

680x0 Stack Frames and Calling Conventions

In general, the 680x0 stack is a grow-down stack that uses two pointers, stored in registers, to access elements on the stack and to manage stack growth. Elements on the stack are 16-bit aligned. Beyond this, it is difficult to make hard and fast statements about the structure of 680x0 stack frames because different calling conventions make different uses of the stack; these differences are enforced by differences in languages (Pascal and C), in language compilers, and in the implementation of system calls.

Pascal and C use stack-based calling conventions but differ in where they store the result and in the order in which they allocate parameters on the stack. Figure 5-4 shows the Pascal and C calling conventions implemented by the MPW compilers. Please consult the documentation provided with your compiler to determine the calling conventions it implements.

Figure 5-4 Pascal and C stack-based calling conventions



The Macintosh Operating System uses a variety of calling conventions. For example, dispatched Toolbox routines use a Pascal stack-based convention but expect a routine selector in register D0 or D1. Operating system routines use register-based calling conventions. For information on the calling conventions used for specific routines, please consult the documentation for the routine in *Inside Macintosh*.

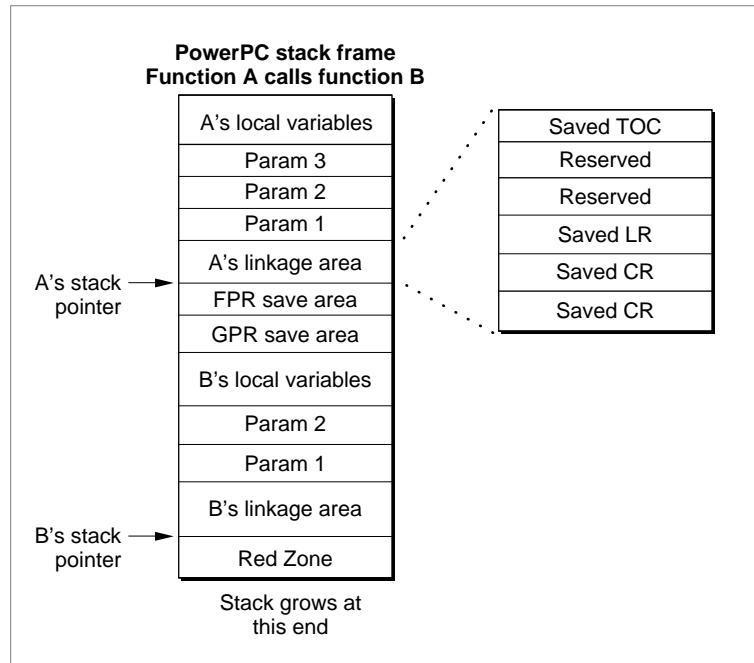
PowerPC Stack Frames and Calling Conventions

The PowerPC stack frame uses register R1 as both a stack pointer and a frame pointer. Once the stack frame is put in place, R1 does not change until another stack frame is created or the current stack frame is deallocated. Parameters are passed in registers, but when you build your application for debugging, they are also copied on the stack from

Low-Level Debugging

left to right. Each field is aligned on a word (32-bit) boundary. Figure 5-5 shows a PowerPC stack frame.

Figure 5-5 The PowerPC stack frame



The PowerPC stack frame, while looking slightly more complicated than the 680x0 stack frame, has the advantage of using one uniform calling convention, which is also used by native Macintosh Operating System routines. The portions of the stack that are used for storing a routine's local variables and parameters vary in size depending on the number and size of the local variables and of the parameters. The FPR and GPR save areas also vary in size, depending on the number of nonvolatile registers that a routine uses. The linkage area is fixed in size and is used in much the same way that the return address and saved A6 fields of the 680x0 stack frame are used—to maintain enough state information to restore the caller's context when the called routine has finished executing. Within the linkage area, the saved Link Register (LR) contains the return address, and the saved TOC allows all routines to have access to their own globals. (For more information about the TOC, see *Inside Macintosh: PowerPC System Software*.)

Although parameters are passed in registers, the stack is used to store copies of these parameters as well as to store excess parameters (if there are more parameters than registers). To learn more about the construction of the PowerPC stack and about the alignment of data on the stack, please see the documentation provided for your compiler and *Assembler for Macintosh With PowerPC*.

System-Level Debugging

The debugger has no mechanism for setting breakpoints on A-traps. All system software routines, however, are called through glue routines. In the browser, the list of source files includes an entry named “**•••synthesized glue•••**”. This contains a list of all glue routines used in your application or library. There is a glue routine for each imported routine. You can set a breakpoint in the glue to catch all calls from your application or library into another library. This allows you to catch all calls into the Toolbox or Operating System.

Debugging Memory-Based Fragments

Usually, the fragments you want to debug are loaded from disk (for example, application fragments). Occasionally, however, you will need to debug a memory-based fragment, such as PowerPC code stored in a resource. Debugging memory-based fragments is slightly more complicated than debugging file-based fragments. You need to make sure that the fragment is registered with the debugger.

The debugger uses the Code Fragment Manager to locate code fragments. With memory-based PowerPC code fragments, however, the Code Fragment Manager removes any information on that fragment from its tables once the code has been prepared. In order to do symbolic debugging on such a fragment, therefore, the fragment must be registered with the debugger. You can do this in two ways:

- In the debugger, Choose General Preferences from the Edit menu and set the option “Stop on code loads for existing processes”.
- On the Power Macintosh computer, hold down the Control key while your application calls the Code Fragment Manager to prepare the fragment for execution.

Once PowerPC code is loaded, you may need to use Map Symbolics to Code (in the File menu) to map the symbolic information file to the code resource. The name you specify depends on how you named the resource. The name of the symbolics file should match the name you passed to the prepare call to the Code Fragment Manager.

If the code does not remain locked in memory, you will not be able to debug it. The debugger does not expect PowerPC code to move once it's been loaded by the Code Fragment Manager.

IMPORTANT

Putting a call to `Debugger` or `DebugStr` in the fragment does not properly register the fragment, because once the fragment has begun execution, the code fragment information the debugger needs has been lost. ▲

Mixed-Mode Debugging

The debugger is able to set breakpoints and step only in PowerPC code. If you try to step through the Mixed Mode Manager and into the emulator, you will eventually reach a point where the Power Macintosh machine just runs and the step is not completed. As a result, before trying to step into the emulator, you should set an appropriate breakpoint.

When stopped in PowerPC code, you can use the Enter MacsBug menu item in the Extras menu to set breakpoints on the 680x0 side. The command `g` (GO) returns you to the stopped point on the PowerPC side. While in MacsBug, you can use the command `es` (Exit to Shell) to terminate your application. In addition, while in MacsBug, the command `ss` (Step Spy) will detect changes to memory only while in 680x0 code. PowerPC code changes will not halt the system.

Debugger Extensions

Contents

About Debugger Extensions	6-3
Writing a Debugger Extension	6-4
Sample Debugger Extension	6-6
Using Callback Routines	6-7
Building a Debugger Extension	6-8
Debugger Extension Reference	6-9
Constants	6-9
Data Structures	6-10
Callback Routines	6-11
Input Routines	6-11
Output Routines	6-13
Utility Routines	6-14

Debugger Extensions

This chapter describes how you can customize and extend the low-level debugger by creating debugger extensions. A **debugger extension** is a C or C++ function that you can call from the debugger by choosing Execute Debugger Extension from the Extras menu and then selecting the name of the extension from the displayed list. Typically, developers use debugger extensions to display complicated data structures that are used to track installed drivers, volumes, open files, VBL tasks, and so on.

Debugger extensions, which are stored in code resources of type 'ndcd', are very similar to MacsBug extensions, which are stored in code resources of type 'dcmd'. The structure of the source code for these two types of resources is nearly identical, and the callback routines used are virtually the same.

This chapter describes

- the structure of 'ndcd' resources
- the callback routines you can use in a debugger extension
- how to build debugger extensions
- how to convert a MacsBug extension into a debugger extension

Note

Debugger extensions are executed on the target machine. ♦

About Debugger Extensions

You create a debugger extension by writing a C function. The structure of the function is described in Listing 6-1 on page 6-5. In addition to C statements, the source code for the debugger extension can include one or more callback routines. These routines allow you to display lines of text, to parse the text entry field, and to read from or write to PowerPC registers.

After compiling the extension and linking it with the appropriate libraries, you need to use the Rez resource compiler to store the executable code in a resource of type 'ndcd'. Then, you must use Rez or a resource editor to add the code resource to the resource fork of the Debugger Nub Preferences file. The debugger loads the extensions in the system heap during system startup. You can use native 'dcmd' resources.

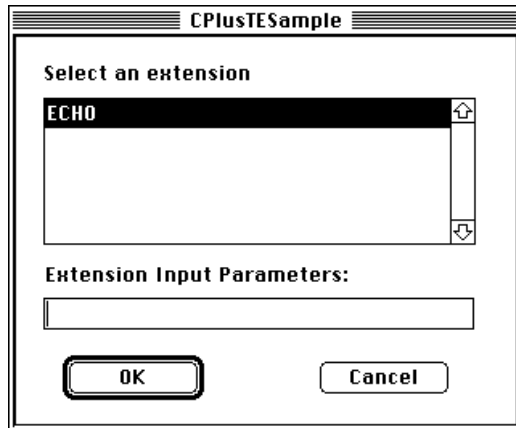
To list available debugger extensions, choose the Execute Debugger Extension item from the Extras menu. The debugger displays a dialog box like the one shown in Figure 6-1. Available debugger extensions are displayed in the scrolling window. If the debugger extension requires that you specify one or more parameters, you can enter these in the text entry field titled Extension Input Parameters. Use a space to separate parameters.

- To execute the extension, click its name, specify the required parameters, and click the OK button or press the Return key.
- To display help for the extension, enter a question mark (?) in the text entry field and press the Return key.

Debugger Extensions

Output and help information from the debugger extension is displayed in the log window.

Figure 6-1 Listing available debugger extensions



Writing a Debugger Extension

Debugger extensions must be PowerPC code and must be written in C or C++. The debugger calls code of type 'ndcd' as a C function declared as follows:

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

When the debugger calls the extension, it passes a single parameter to the function, a pointer to a parameter block. The parameter block passed to the `CommandEntry` routine is a structure used to store information the routine needs. It is declared as follows:

```
typedef struct {
    long      *registerFile;
    short     request;
    Boolean   aborted;
} extensionBlock;
```

Debugger extensions use only the `request` field of the structure. The `registerFile` and `aborted` fields are retained as part of the structure to ensure compatibility with MacsBug extensions.

Debugger Extensions

Note

MacsBug extensions use the `registerFile` field as a pointer to an array containing the contents of the 680x0 registers. The debugger ignores this field. To obtain information about the PowerPC registers or to change the value of one of the PowerPC registers, you can use the callback routines `dcmdReadRegister` and `dcmdWriteRegister`, described in “Utility Routines” on page 6-14. ♦

The `request` field contains one of the constants listed in Table 6-1, which the debugger passes to the extension:

Table 6-1 request field values

<code>dcmdInit</code>	The value of the <code>request</code> field when the debugger first calls the extension. The call is made when the debugger is loaded into memory. In response to this request, the extension can perform any necessary initialization actions. For example, it can initialize its global variables or gather information about the operating environment. Your debugger extension can also ignore this constant.
<code>dcmdDoIt</code>	The value of the <code>request</code> field when the user executes the command.
<code>dcmdHelp</code>	The value of the <code>request</code> field when the user asks for help by selecting the extension and entering a question mark (?) in the text entry field.

Listing 6-1 shows the format of the source code for a debugger extension. A debugger extension can contain more than one procedure; however, the main procedure must be called `CommandEntry`.

Listing 6-1 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 passes control to one of
    three functions: dcmdInit, dcmdHelp, or dcmdDoIt.*/

    switch (paramPtr->request){
        case dcmdInit:
            /*code that executes at initialization time*/
```

Debugger Extensions

```

        case dcmdHelp:
            /*code that displays syntax and help info*/
        case dcmdDoIt:
            /*code that performs command's normal function*/
        }
    }

```

Sample Debugger Extension

Listing 6-2 shows sample source code for the Echo debugger extension. This command echoes extension parameters; if the parameter is an expression, it evaluates the expression. In addition to C statements, the sample code uses debugger callback routines. These are introduced in the next section, “Using Callback Routines.”

Listing 6-2 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:
            break;

        case dcmdDoIt:
            do {
                /*save position so we can rewind if error*/

```

Debugger Extensions

```

        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

Besides using C or C++ statements and directives, you can use callback routines in writing debugger extensions. Callback routines allow you to display lines of text and parse the text entry field. They are called **callback routines** because the debugger calls the debugger extension, which calls back to the debugger to use the routine. Most of the routine declarations are found in the file `dcmd.h`. The declarations of the routines `dcmdReadRegister` and `dcmdWriteRegister` are found in the file `ndcd.h`. The implementation of all the callback routines is found in the library file `NubExt.xcoff`.

You use debugger callback routines to

- parse the text entry field. These routines are described in the section “Input Routines” on page 6-11.
- display help text or other output. These routines are described in the section “Output Routines” on page 6-13.
- obtain or change the values of PowerPC registers. These routines are described in the section “Utility Routines” on page 6-14.
- obtain a trap name. This routine is described in the section “Utility Routines” on page 6-14.

Note

The routines `dcmdGetBreakMessage`, `dcmdGetNameAndOffset`, `dcmdGetMacroName`, `dcmdSwapWorlds`, `dmcdSwapScreens`, `dcmdScroll`, and `dcmdDrawPrompt` are stub routines included for compatibility with MacsBug extensions. ♦

Building a Debugger Extension

To build a debugger extension and to place it in the debugger's preferences file, 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 `NubExt.xcoff` file.**
3. **Convert the linked `.xcoff` file into a PEF file using `MakePef`. (Newer versions of the linker produce PEF files directly and don't require this step.)**
4. **Use Rez to create an 'ndcd' code type resource.**
5. **Use Rez or a resource editor to add the resource to the resource fork of the Debugger Nub Preferences file.**
6. **Reboot the machine.**

The new debugger extension will be displayed when you choose Execute Debugger Extensions from the Extras menu.

Table 6-2 lists the files you need to build a debugger extension.

Table 6-2 Header and library files for debugger extensions

File	Use
<code>Put.c</code>	Formatting routines used by debugger extensions. This file is optional.
<code>Put.h</code>	Header file for formatting routines.
<code>dcmd.h</code>	Header file for debugger extensions. This file does not include declarations for <code>dcmdReadRegister</code> and <code>dcmdWriteRegister</code> routines.
<code>ndcd.h</code>	Header file for debugger extension. This file includes declarations for <code>dcmdReadRegister</code> and <code>dcmdWriteRegister</code> routines.
<code>ndcdGlue.o</code>	Standard glue file you must link with.
<code>NubExt.xcoff</code>	Library file containing the implementation of the routines declared in <code>dcmd.h</code> and <code>ndcd.h</code> .

Listing 6-3 shows the makefile used for the Echo extension.

Listing 6-3 Debugger extension makefile

```
TARGETNAME = Echo
CMNDIR = ':::'
OBJECTS = Echo.o {CMNDIR}ndcdGlue.o {CMNDIR}NubExt.xcoff
```

Debugger Extensions

```

COMPILE = PPCC -w -appleext on -I {CMNDIR}

{TARGETNAME} ff Echo.makefile {OBJECTS}
    PPCLink -main ndcdExport {OBJECTS} 0
    "{MPW}Libraries:PPCLibraries:StdCLib.xcoff" 0
    -o {TARGETNAME}.xcoff
    makepef -e ndcdExport -l NubExt.xcoff=NativeNub
    -l StdCLib.xcoff=StdCLib
    -o {TARGETNAME}.pef {TARGETNAME}.xcoff    # create pef executable
    rez EchoAdd.r -a -o {TARGETNAME}          # rez it as an 'ndcd' resource
    setfile -t 'rsrc' -c 'RSED' {TARGETNAME}  # set the file type & creator

Echo.o f Echo.makefile Echo.c
{COMPILE} Echo.c -o Echo.o

```

Listing 6-4 shows the resource description file `EchoAdd.r`. The `read` statement defines the 'ndcd' resource (by specifying its ID and type, and setting its attributes) and reads the data fork of the `Echo.pef` file as the data for the resource.

Listing 6-4 Sample resource description file to create an 'ndcd' resource

```

// EchoAdd.r file

read 'ndcd'(101,"Echo",preload,sysheap,locked)"Echo.pef";

```

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:

Debugger Extensions

```

#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:

Debugger Extensions

```
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.
request	A field with a value of <code>dcmdInit</code> , <code>dcmdDoIt</code> , or <code>dcmdHelp</code> . In response to <code>dcmdInit</code> , the extension should perform any required initialization. In response to <code>dcmdDoIt</code> , the extension should execute its main function. In response to <code>dcmdHelp</code> , 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 and for obtaining a trap name.

Note

The routines `dcmdGetBreakMessage`, `dcmdGetNameAndOffset`, `dcmdGetMacroName`, `dcmdSwapWorlds`, `dcmdSwapScreens`, `dcmdScroll`, and `dcmdDrawPrompt` are stub routines included for compatibility with MacsBug extensions. The declarations of these routines are found in the `dcmd.h` file. ♦

Input Routines

You can use the following routines to parse user input.

dcmdGetPosition

The `dcmdGetPosition` routine returns a short integer specifying the current command line position.

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

dcmdSetPosition

The `dcmdSetPosition` routine 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

The `dcmdGetNextChar` routine returns the next character or a return character if the entire line has been scanned.

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

dcmdPeekAtNextChar

The `dcmdPeekAtNextChar` routine 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

The `dcmdGetNextParameter` routine 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

The `dcmdGetNextExpression` routine 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

The `dcmdDrawLine` routine 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

The `dcmdDrawString` routine 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

The `dcmdDrawText` routine 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 and to obtain a trap name.

dcmdGetTrapName

The `dcmdGetTrapName` routine returns the trap name for the specified trap number. If the name can't be found, the routine returns an empty string.

```
pascal void dcmdGetTrapName(short trapNumber, Str255 trapName);
```

`trapNumber` A hexadecimal integer specifying the trap number.

`trapName` On exit, the name of the trap specified by `trapNumber`.

dcmdReadRegister

The `dcmdReadRegister` routine 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" on page 6-9 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

The `dcmdWriteRegister` routine writes a value to a PowerPC register.

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

<code>regNum</code>	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” on page 6-9 for more information.
<code>regSize</code>	An integer specifying the size of the register: 32 specifies a general purpose register; 64 specifies a floating-point register.
<code>regValue</code>	The value you want to place in the register.

Expression Evaluation

Contents

Expression Grammar	A-3
Symbols	A-3
Constants	A-3
Register Names	A-3
Precedence of Operators	A-4
Expression Syntax	A-5
Error Messages	A-9

Expression Evaluation

This appendix describes the grammar of C and C++ expressions that can be evaluated using the debugger's expression evaluator and explains the meaning of the error messages returned by the evaluator.

Expression Grammar

The grammar supported by the expression evaluator is a union of C and C++ grammar, not including assignments and member pointers. The evaluator also supports machine register access as described in the section "Register Names."

IMPORTANT

The evaluator does not support type casting. However, you can use the Evaluate item from the Evaluate menu to evaluate an expression using a data type you specify. For the same reason, the evaluator cannot handle expressions of the form `sizeof (typeName)`, although it can handle expressions of the form `sizeof (variableName)`. For additional information, see "Names, Addresses, or Expressions" on page 3-28. ▲

Symbols

A symbol is a character or a combination of characters used to represent an identifier, a numeric constant, or a string.

Constants

The expression evaluator handles constants according to the rules defined in ANSI C and Kernighan and Ritchie. In addition, it scans

- a number (0–9) preceded by a pound sign (#) as a decimal number
- a number (0–9, A–F) preceded by a dollar sign (\$) as a hexadecimal number

The value of the specified number determines its type as long, unsigned long, or double.

Register Names

Register names have predefined meaning in the debugger, as shown in Table A-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 (Δ) (Option-J) symbol before the register name. 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
```

Expression Evaluation

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

$$x + \Delta R1$$

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

Note

You can speed up processing by using the Δ prefix in register names, whether or not you have also declared those names to be variables. ♦

Table A-1 lists the PowerPC register names.

Table A-1 PowerPC register names

Register name	Use
R0–R31	General-purpose register
FP0–FP31	Floating-point register
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
FPSCR	Floating-Point Status and Control Register

Table A-2 lists the 680x0 register names.

Table A-2 680x0 register names

Register name	Use
A0–A7	Address registers
D0–D7	Data registers
PC	Program counter

Precedence of Operators

Table A-3 describes the precedence of operators in the evaluation of expressions. For each type of operator (unary or binary), the operators are listed in order of precedence. Operators having equal precedence are listed on the same line.

Expression Evaluation

Table A-3 Precedence of operators

Operator type	Operator	Name of operation
Unary	~	Bitwise NOT (complement)
	-	Arithmetic negation (unary + also allowed)
	* ->	Indirection
	&	Address of
	sizeof	Size of
	. ::	Structure field
Binary	* / %	Multiplication, division, modulus
	+ -	Addition, subtraction
	<< >>	Left shift, right shift
	< > <= >=	Less than, greater than, less than or equal, greater than or equal
	== !=	Equality, inequality
	&	Bitwise AND
	^	Bitwise XOR (exclusive disjunction)
		Bitwise OR (inclusive disjunction)
	&&	Logical AND
		Logical OR
	?:	Conditional

Expression Syntax

An expression may itself be produced by a combination of other expressions. This section describes the possible combinations and the syntax of expressions. It does not present a complete C and C++ grammar. It is only concerned with the valid form of expressions recognized by the debugger.

expression:

*/*empty*/
conditionalExpression
unaryExpression*

conditionalExpression:

*logicalOrExpression
logicalOrExpression ? conditionalExpression : conditionalExpression*

logicalOrExpression:

*inclusiveAndExpression
logicalOrExpression || logicalAndExpression*

logicalAndExpression:

*inclusiveOrExpression
logicalAndExpression && inclusiveOrExpression*

Expression Evaluation

inclusiveOrExpression:

exclusiveOrExpression
inclusiveOrExpression | *exclusiveOrExpression*

exclusiveOrExpression:

andExpression
exclusiveOrExpression ^ *andExpression*

andExpression:

equalityExpression
andExpression & *equalityExpression*

equalityExpression:

relationalExpression
equalityExpression == *relationalExpression*
equalityExpression != *relationalExpression*

relationalExpression:

shiftExpression
relationalExpression < *shiftExpression*
relationalExpression > *shiftExpression*
relationalExpression <= *shiftExpression*
relationalExpression =< *shiftExpression*

shiftExpression:

additiveExpression
shiftExpression << *additiveExpression*
shiftExpression >> *additiveExpression*

additiveExpression:

multiplicativeExpression
additiveExpression + *multiplicativeExpression*
additiveExpression - *multiplicativeExpression*

multiplicativeExpression:

pmExpression
multiplicativeExpression * *castExpression*
multiplicativeExpression / *castExpression*
multiplicativeExpression % *castExpression*

castExpression:

unaryExpression
(*simpleTypeName*) *castExpression*

unaryExpression:

postfixExpression
& *castExpression*
* *castExpression*
+ *castExpression*
- *castExpression*
~ *castExpression*
! *castExpression*
sizeof *castExpression*

postfixExpression:

primaryExpression
postfixExpression [*conditionalExpression*]

Expression Evaluation

```

    postfixExpression ( conditionalExpression )
    simpleTypeName ( conditionalExpression )
    postfixExpression . memberName
    postfixExpression -> memberName
    postfixExpression :: classMemberName

primaryExpression:
    primaryExpression
    ( conditionalExpression )
    name

literal:
    IntegerLiteral
    charLiteral
    floatConstant
    stringLiteral

memberName:
    identifier
    qualifiedName

name:
    IDENTIFIER
    qualifiedName

qualifiedName:
    className IDENTIFIER
    className TYPE_NAME

identifier:
    IDENTIFIER
    TYPE_NAME
    regName

className:
    CLASS_NAME
    className CLASS_NAME

simpleTypeName:
    TYPE_NAME
    completeClassName
    qualifiedTypeName
    basicType

completeClassName:
    className
    className :: qualifiedClassName

qualifiedClassName:
    className
    className :: qualifiedClassName

qualifiedTypeName:
    typedefName
    className :: qualifiedTypeName

basicType:
    VOID
    CHAR

```

Expression Evaluation

SHORT
INT
LONG
SIGNED
UNSIGNED
FLOAT
DOUBLE
EXTENDED
COMP

regName:

reg
Δreg

reg:

A PowerPC or 68K register name, as shown in Table A-1 and Table A-2 on page A-4.

IntegerLiteral:

hexLiteral
hexLiteral U
hexLiteral u
hexLiteral L
hexLiteral l
decLiteral
decLiteral U
decLiteral u
decLiteral L
decLiteral l

hexLiteral:

0x *hexNum*
0X *hexNum*
\$*hexNum*

hexNum:

hexDigit hexNum
hexDigit

hexDigit:

decDigit
A
B
C
D
E
F

decLiteral:

decNum
#*decNum*

decNum:

decDigit decNum
decDigit

Expression Evaluation

decDigit: 0 ... 9

charLiteral: 'CHAR_CONSTANT'

stringLiteral: "CHAR_CONSTANT"

Error Messages

The following error messages are returned by the debugger when evaluating expressions.

Syntax Error

Semantic error. Check operand's type.

Variable is a variable that either is not defined in your program or cannot be found.

Not enough memory to complete the expression evaluation

Expression is too long. It cannot be evaluated

The variable following the '.' is not a struct/union member.

Variable is not a struct/union.

Cannot evaluate this expression. You might have used syntax that is not part of the grammar.

Can only dereference Pointers and Vectors.

Failed to do the type conversion. Check the operand's type.

Register is a register that cannot be accessed.

Evaluate what???

The string following the 'Δ' must be a valid register name.

The string following the '?' must be a valid hex number.

The string following the '#' must be a valid decimal number.

Debugger Menu Reference

Contents

File Menu	B-3
Open...	B-3
Open ROM map	B-3
Map Symbolics to Code	B-3
Close	B-4
Save Log As...	B-4
Save Performance	B-4
Page Setup...	B-4
Print Window...	B-4
Quit	B-4
Edit Menu	B-5
Undo	B-5
Cut	B-5
Copy	B-5
Paste	B-6
Clear	B-6
Select All	B-6
Show Clipboard	B-6
General Preferences...	B-6
Target Preferences...	B-8
Symbolic Mapping Preferences...	B-9
Remember Window Position	B-11
Views Menu	B-11
Show Stack Crawl	B-11
Show Globals	B-13
Show Registers	B-14
Show FPU Registers	B-15
Show Memory	B-17
Show Instructions	B-18
Show 68K Instructions	B-19

Show Tasks	B-19
Show Fragment Info	B-20
Find Code For...	B-22
Find Code For ""	B-22
Source File For ""	B-22
Show Log Window	B-22
Control Menu	B-23
Run ""	B-24
Stop ""	B-24
Kill ""	B-24
Resume ""	B-24
Propagate Exception	B-24
Target ""	B-25
Untarget ""	B-25
Launch	B-25
Step	B-25
Step Into	B-25
Step Out	B-26
Step To Branch	B-26
Step To Branch Taken	B-26
Animate	B-26
Show Breakpoints List	B-26
Clear All Breakpoints	B-27
Show/Hide Control Palette	B-27
Evaluate Menu	B-27
Show Evaluation Window	B-28
Evaluate ""	B-28
Evaluate "this"	B-29
Displaying Values in Different Formats	B-29
Performance Menu	B-29
New Session	B-30
Show Source	B-31
Configure Utility	B-31
Configure Report	B-32
Enable / Disable Utility	B-33
Gather Report	B-33
The Summary View	B-34
The Statistics View	B-35
Extras Menu	B-36
Show PC	B-36
Search Memory	B-36
Enter MacsBug	B-37
Stop For DebugStrs	B-37
Execute Debugger Extension...	B-37
Snapshot Active Window	B-38
Windows Menu	B-38

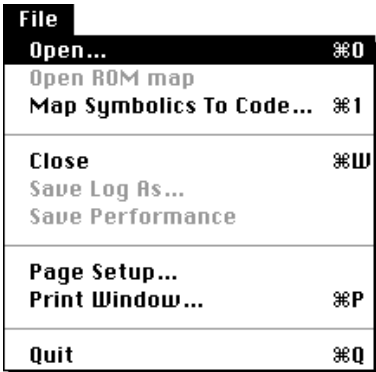
Debugger Menu Reference

This appendix describes the debugger menus. The menus are described in the order of their appearance in the menu bar: File, Edit, Views, Control, Evaluate, Performance, Extras, and Windows.

File Menu

You use commands in the File menu to open or close symbolic information files, to save the contents of log files, to print files, and to quit the debugger. Figure B-1 shows the File menu. The following subsections describe File menu items in the order of their appearance.

Figure B-1 The File menu



Open...

Choose Open from the File menu (or press Command-O) to open a symbolic information file. The debugger displays a dialog box containing a scrollable window from which you can select the file to open. You can choose Open more than once to open more than one symbolic information file.

Open ROM map

To activate this command, choose General Preferences from the Edit menu and select Use ROM Map in the dialog box that appears. Choosing this menu item brings up a standard file dialog box that allows you to select a ROM map for the debugger to use.

Map Symbolics to Code

Not available for 68K Mac Debugger. Power Mac Debugger maps executable code to symbolic files by name. For applications, the name of the symbolic file should be the name of the application with `.xcoff` or `.xSYM` appended. For example, the application

Debugger Menu Reference

Sample automatically maps to the symbolics file `Sample.xcoff` (or `Sample.xSYM`). For shared libraries, the name in the symbolics file should be the name in the 'cfrg' resource with `.xcoff` or `.xSYM` appended. If the application or shared library symbolics file does not have the same name, then the debugger is not able to automatically map the symbolics to the executable code, and you must use the Map Symbolics to Code command.

Close

Choose Close from the File menu (or press Command-W) to close the frontmost window. Note that when you close a browser window, all of the symbolic information for that symbolic information file is removed from any other window associated with it, for example, the stack window or the instruction window.

Save Log As...

Choose Save Log As from the File menu to save the contents of the log window to an MPW text file. The debugger writes the output of `DebugStr` calls to the log window.

Save Performance

Not available for 68K Mac Debugger. Choose Save Performance from the File menu to save a performance document. The document is saved as a tab-delimited MPW text file. You can view the file using MPW or a spreadsheet program such as Microsoft Excel.

Page Setup...

Choose Page Setup from the File menu to set up page formatting for printing. A dialog box is displayed that you can use to specify the desired settings.

Print Window...

Choose Print Window from the File menu (or press Command-P) to print the contents of a window. Whether the debugger prints just the visible contents of the window or the entire scrollable region of the window depends on the window being printed:

- For the memory and instruction windows, which scroll over the entire address space, only the visible portion of the window is printed.
- For code views, all the source code or assembly code for the selected function is printed.

Quit

Choose Quit from the File menu (or press Command-Q) to quit the debugger. If the program you are debugging is currently stopped, the debugger displays a dialog box

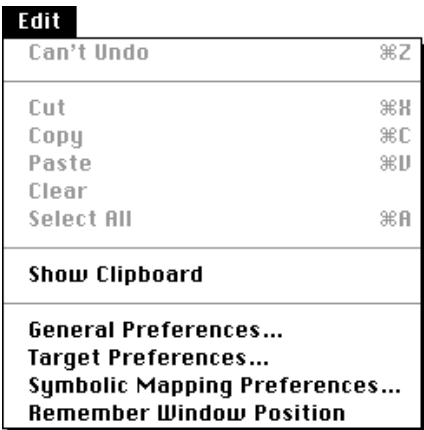
Debugger Menu Reference

asking if you would like to resume execution of the target application or quit it. In most cases, you should resume execution of the target application before quitting the debugger.

Edit Menu

You use items in the Edit menu to undo changes and to redefine the debugger’s default preferences settings. The cut and paste operations are provided mainly for the benefit of desk accessories and for cutting and pasting values (addresses, register contents, expressions) from one debugger window to another. Figure B-2 shows the Edit menu. The following subsections describe Edit menu items in the order of their appearance.

Figure B-2 The Edit menu



Edit	
Can't Undo	⌘Z
Cut	⌘H
Copy	⌘C
Paste	⌘U
Clear	
Select All	⌘A
Show Clipboard	
General Preferences...	
Target Preferences...	
Symbolic Mapping Preferences...	
Remember Window Position	

Undo

Choose Undo from the Edit menu (or press Command-Z) to undo the last change. For example, you can use this command to undo a change to memory or to a register value. If the last change can’t be undone, the command reads Can’t Undo.

Cut

Choose Cut from the Edit menu (or press Command-X) to cut a selection. You can then paste the selection using the Paste command.

Copy

Choose Copy from the Edit menu (or press Command-C) to copy a selection. You can then paste the selection using the Paste command.

Debugger Menu Reference

Paste

Choose Paste from the Edit menu (or press Command-V) to paste a selection.

Clear

Choose Clear from the Edit menu to clear the Clipboard.

Select All

Choose Select All from the Edit menu (or press Command-A) to select the contents of the current window.

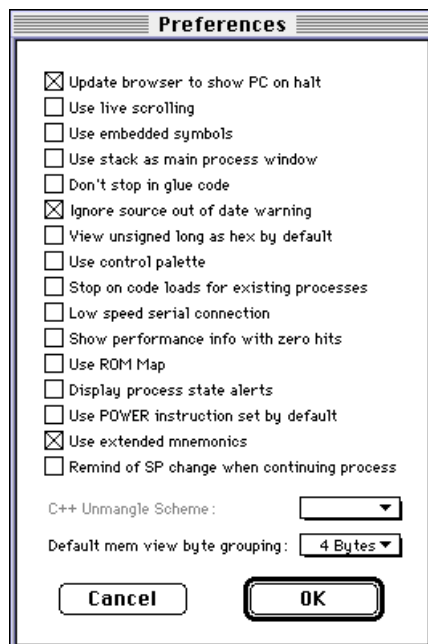
Show Clipboard

Choose Show Clipboard from the Edit menu to display the Clipboard file.

General Preferences...

Choose General Preferences from the Edit menu to display the General Preferences dialog box. The debugger uses the values you select in the dialog box to redefine some features of the interface and of the debugger's memory display. Figure B-3 shows the General Preferences dialog box.

Figure B-3 The General Preferences dialog box



Debugger Menu Reference

Table B-1 presents information about the General Preferences options.

Table B-1 General Preferences options

Option	Effect when selected
Update browser to show PC on halt	Updates the source code display in the browser so that the current program counter is displayed whenever the program is stopped.
Use live scrolling	Scrolls windows as the scroll box moves in the scroll bar. Otherwise, scrolling occurs when the scroll box stops moving.
Use embedded symbols	Displays function name symbols in the stack and instruction windows when no symbolic information file is found. 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 down the debugger.
Use stack as main process window	Uses the stack window rather than the registers window as the main process window. The debugger must have an active, unclosable window at all times. If you do not select this option, the debugger uses the registers window as this “main process” window by default. You must relaunch the debugger to have this option take effect.
Don’t stop in glue code	The debugger stops only in code that has symbolic information mapped to it. The debugger doesn’t stop in glue code.
Ignore source out of date warning	Eliminates “Source out of date” warnings. The debugger compares modification dates to determine if a symbolic information file is out of date with respect to the target application or the source files. For example, if you recompile the executable file without recreating the symbolic information file or if you delete comments from one of your source files, the debugger alerts you to the fact that the files’ modification dates no longer match. You can safely ignore this message if you know that whatever changes have been made will not affect the mapping between files.
View unsigned long as hex by default	Displays unsigned long values as hexadecimal values in the stack crawl, variable, and global variables windows. Leaving this option unselected displays unsigned long values as unsigned decimal values.
Use control palette	Causes the control palette to be displayed when the debugger is launched.
Stop on code loads for existing processes	Stops a targeted process when it loads any code, for example, when an application loads a code fragment.
Low speed serial connection	<i>Not available for 68K Mac Debugger.</i> Check this box if your host machine is slower than a Macintosh Quadra and you are experiencing communication problems with the target machine. If you check this box, you must also check the low-speed serial connection box in the secondary configuration window of the Debugger Nub Controls dialog box.
Show performance info with zero hits	<i>Not available for 68K Mac Debugger.</i> Instructs the Adaptive Sampling Profiler (ASP) to display buckets with zero hits in a performance report. Leave this option unselected to suppress the display of all empty buckets.

Debugger Menu Reference

Table B-1 General Preferences options (continued)

Option	Effect when selected
Use ROM Map	When you check this box, the debugger uses a ROM map. The first time the nub tries to connect to the host, the debugger requests the location of the ROM map through a standard file dialog box. The debugger then saves the location of the ROM map file and opens it automatically on subsequent launches of the host.
Display process state alerts	Check this box to display all informational alerts for a process. When you do not check the box, then some of the alerts are not displayed; for example, no alert appears when the debugger stops due to a break on launch.
Use POWER instruction set by default	<i>Not available for 68K Mac Debugger.</i> Displays POWER mnemonics in the instruction window. If you leave this box unchecked, the debugger displays PowerPC mnemonics.
Use extended mnemonics	<i>Not available for 68K Mac Debugger.</i> When you check this box, Power Mac Debugger disassembles PowerPC code using the extended mnemonic set. (If you selected the Use POWER instruction set by default option, then this preference does nothing.) If you do not check this box, then the debugger does not use extended mnemonics.
Remind of SP change when continuing process	<i>Not available for 68K Mac Debugger.</i> You can change the stack pointer in order to stack crawl from any location. If you then step or run the process, a crash is likely to occur. If this box is checked, the debugger reminds you of this and gives you the option to reset the stack pointer to the original value before stepping or running the process.
C++ Unmangle Scheme	Use this option to select a custom C++ unmangle scheme that you have previously added to the Macintosh Debugger Preferences folder. Refer to Appendix D, “Creating Custom Unmangle Schemes,” for more information on unmangle schemes.
Default mem view byte grouping	Sets the default memory display. Choose a value from the pop-up menu to set the default memory display at 4-byte grouping, 2-byte grouping, or 1-byte grouping.

Target Preferences...

Choose Target Preferences from the Edit menu to specify whether your default choice is one-machine or two-machine debugging. If you want to make this choice at startup, select “Ask on startup”.

There are two different dialog boxes for this option, one for Power Mac Debugger (shown in Figure B-4), and one for 68K Mac Debugger (shown in Figure B-5).

Select Local for one-machine debugging.

For remote (two-machine) debugging with 68K Mac Debugger, select AppleTalk.

For remote (two-machine) debugging with Power Mac Debugger, select the port to be used for the connection: AppleTalk, Modem port, or Printer port. Use AppleTalk for source-level debugging and the Modem or Printer port for low-level debugging with a serial cable.

Debugger Menu Reference

Changes made at this screen do not take effect until relaunch.

Figure B-4 Target preferences for Power Mac Debugger

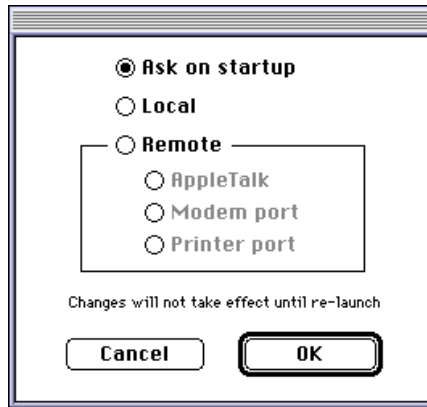
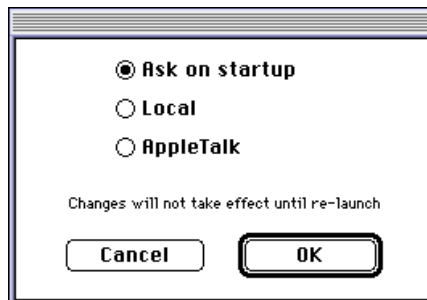


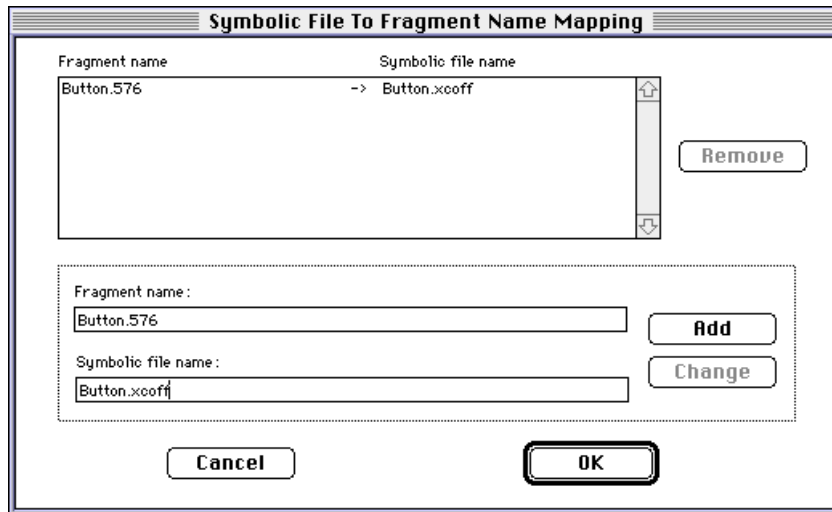
Figure B-5 Target preferences for 68K Mac Debugger



Symbolic Mapping Preferences...

Choose Symbolic Mapping Preferences from the Edit menu when you need to inform the debugger of the name of the symbolic file to use for a particular fragment that you are debugging. The debugger displays a dialog box like the one shown in Figure B-6.

Debugger Menu Reference

Figure B-6 The Symbolic File to Fragment Name Mapping dialog box

For Power Mac Debugger, symbolic file mapping might be needed when you have more than one fragment in a PEF container. When the nub notifies the host of loads, it appends an offset to the end of the fragment name, for example, `Button.576`. You could use the Map Symbolics to Code command (File menu) each time to inform the debugger of the offset. The Symbolic Mapping Preferences command allows you to set this up once so that it is done automatically thereafter (assuming the offset does not change). The next time you open `Button.xcoff`, the Browser's name will be `Button.576` instead of just `Button`. When the fragment is loaded within the targeted application, it is now automatically mapped to the symbolics in the browser.

For 68K Mac Debugger, one case requiring symbolic file mapping occurs when you are debugging a fat shared library or drop-in. A fat shared library or drop-in is created with the MergeFragment tool, which places both the PowerPC and 680x0 fragments in the data fork of the shared library. To distinguish fragment names, the fragment that appears after the first fragment in the shared library has its offset appended to its name. For example, a shared library named `Button` is created by merging the PowerPC and 680x0 fragments with the following command:

```
MergeFragment -x Button.ppc Button.68k Button
```

This places the 680x0 fragment after the PowerPC fragment in the shared library called `Button`, and the 680x0 fragment name will be something similar to `'Button.576'`. In order to debug this fragment, you must explicitly tell the debugger to match the `Button.576` fragment to the symbolic file named `Button.NJ` by using the Symbolic Mapping Preferences menu item.

Remember Window Position

Choose Remember Window Position from the Edit menu to save the location and size of the active window in the preferences file. The window is saved by window type only, not by instantiation. When you open multiple windows of the same type, they come up in a staggered position. This option also remembers the size and position of any panes within windows that have them, such as browser, stack, and task windows.

Views Menu

You use commands in the Views menu to display selected areas of memory, to navigate through source code, and to display the log window. Figure B-7 shows the Views menu. The following subsections describe Views menu items in the order of their appearance.

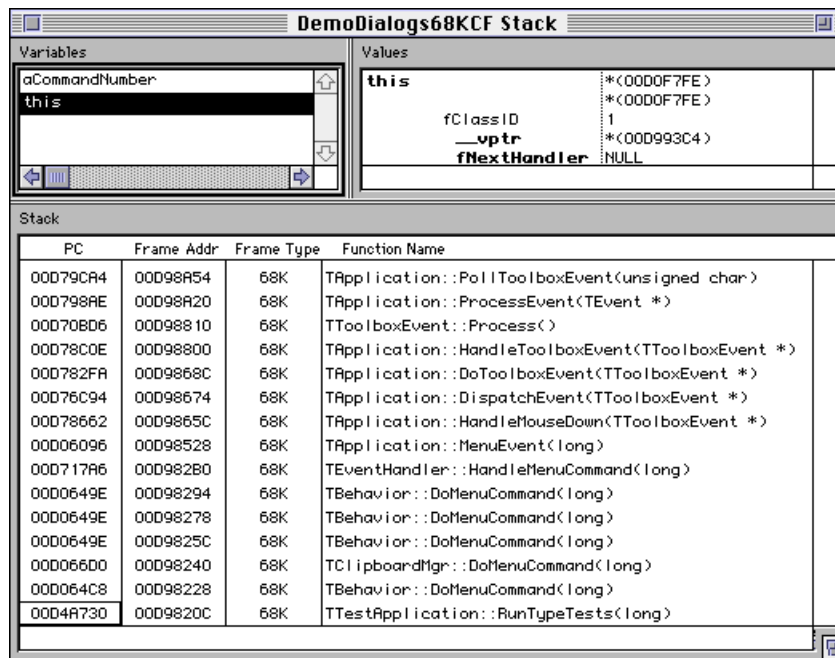
Figure B-7 The Views menu

Views	
Show Stack Crawl	⌘J
Show Globals	⌘L
Show Registers	⌘K
Show FPU Registers	
Show Memory	⌘M
Show Instructions	⌘D
Show 68K Instructions	⌘8
Show Tasks	
Show Fragment Info	
Find Code For...	⌘F
Find Code For <small>“ ”</small>	⌘H
Source File for <small>“ ”</small>	
Show Log Window	

Show Stack Crawl

Choose Show Stack Crawl from the Views menu (or press Command-J) to display the stack. Figure B-8 shows a sample stack display.

Debugger Menu Reference

Figure B-8 The stack display

The stack window is composed of three panes. The upper-left pane displays the local variables for the stack frame selected in the lower pane. The upper-right pane displays the value of the variable selected in the upper-left pane. The lower pane is a stack crawl window. You can resize the top panes by placing the cursor on the split lines separating the panes and pressing to display the horizontal resize icon. When the icon is displayed, drag to the left or right to resize the panes.

For each stack frame in the lower pane, the following information is displayed: the program counter, the stack pointer, the type of frame, and the name of the routine (if the name is known.) If the name is not known, it is indicated by a series of question marks (????). The last stack frame shown is the one at the top of stack—that is, it is the frame for the current function. Remember that the stack grows toward low memory.

If you select a stack frame in the stack crawl window, the debugger displays the local variables for that frame in the upper-left pane. If you select one of those variables, the debugger displays its value in the upper-right pane. If the variable is a structured data type, the debugger displays the value of each of its fields. Special notation is used to designate addresses, pointers, and handles in the upper-right pane.

- A value in parentheses specifies an address.
- An address preceded by an asterisk (*) specifies a pointer.

You can access an instruction, memory, or browser display directly from the stack window by double-clicking selected items in the lower pane:

Debugger Menu Reference

- If you double-click a PC address, the debugger displays an instruction window, with disassembly beginning at the PC address.
- If you double-click a frame address, the debugger displays a memory window, with the memory dump beginning at the frame address.
- If you double-click a function name, the debugger displays a browser window or an instruction window.

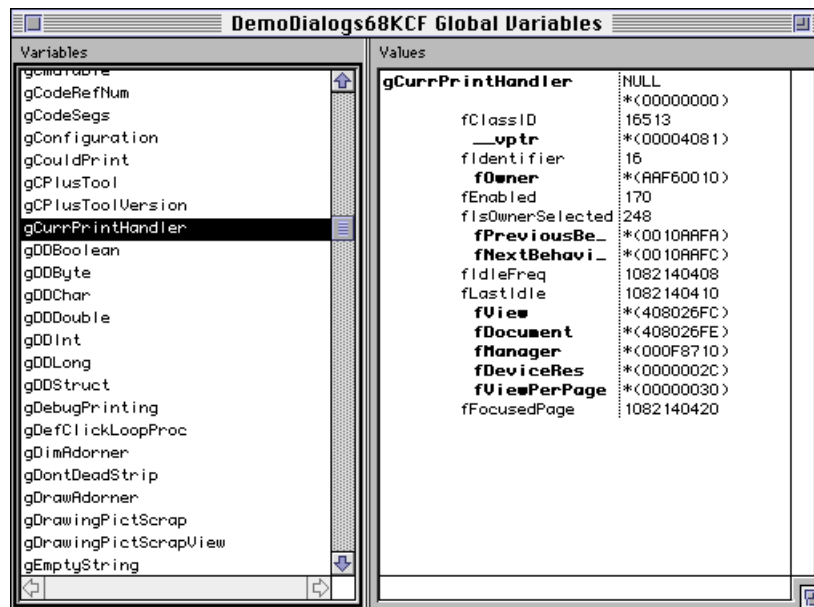
To evaluate a local variable in a recursive routine, select the frame that you are interested in. The local variables for that frame are displayed in the upper-left pane as usual.

For additional information about the structure of stack frames, see Chapter 5, “Low-Level Debugging.”

Show Globals

Choose Show Globals from the Views menu (or press Command-L) to display the addresses, names, and values of your program’s global variables. Figure B-9 shows a sample display of the global variables window.

Figure B-9 The global variables display window



The global variables window is composed of two panes. The left pane lists your program’s global variables. If multiple symbolic files are open, the source file name will also be shown to the right of the global variable name. To display the value of a variable, select the variable’s name. Its value is shown in the right pane. If the variable is a structured data type, the value of each field is also shown in the right pane.

Debugger Menu Reference

Special notation is used to designate addresses, pointers, and handles in the right pane.

- A value in parentheses specifies an address.
- An address preceded by an asterisk (*) specifies a pointer.

You can change the value of a global variable by selecting its value in the right pane and editing it. Press Tab or Enter to make the change.

You can resize the panes by placing the cursor on the vertical lines (split lines) separating the panes and pressing to display the horizontal resize icon. Then drag to the right or left to resize the panes.

Show Registers

Choose Show Registers from the Views menu (or press Command-K) to display the registers window. The registers window allows you to display or change the values of the general-purpose registers and is different for Power Mac Debugger and 68K Mac Debugger. Figure B-10 shows a sample PowerPC registers display.

Figure B-10 The PowerPC registers display window

Muslin Registers																		
		CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7									
PC		0058D3C8	CR	0	1	0	0	0	1	0	0000	0000	0000	0000	0100	1000		
LR		0058D3C8	< > = 0 X E U 0															
CTR		409E0DF0	S	0	C	Compare				Count								
			XER	0	0	0	00				00							
R 00	000CB648	R 08	6808D434	R 16	00000000	R 24	00000000											
SP	005CA230	R 09	00000010	R 17	00000000	R 25	001388E2											
T0C	00011240	R 10	00020B20	R 18	00000000	R 26	00000001											
R 03	0059403F	R 11	00000000	R 19	00000000	R 27	00000001											
R 04	00033022	R 12	0058D3C8	R 20	00000000	R 28	005CA448											
R 05	FFFFFF1AD	R 13	00000000	R 21	00000000	R 29	0059853C											
R 06	636E666E	R 14	00000000	R 22	00000000	R 30	00598530											
R 07	005CA0E0	R 15	00000000	R 23	00000000	R 31	00594038											

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

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

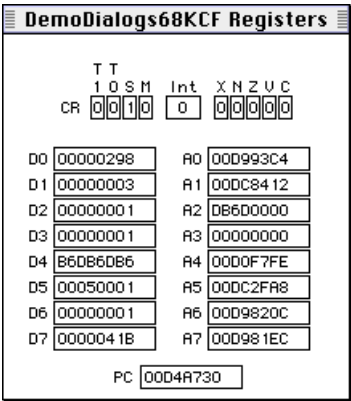
Debugger Menu Reference

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). ♦

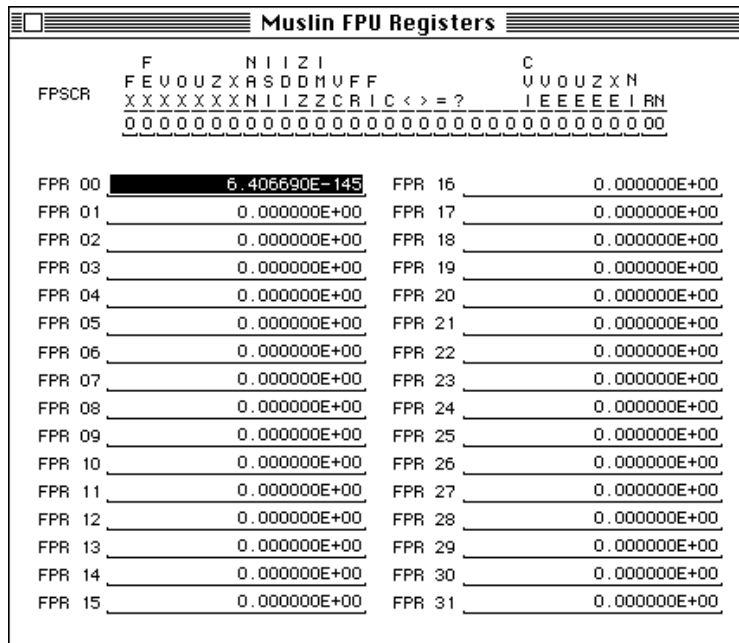
Figure B-11 shows a sample 680x0 registers window.

Figure B-11 The 680x0 registers window



Show FPU Registers

To display the PowerPC floating-point registers, choose Show FPU Registers from the Views menu. The debugger displays a window like the one shown in Figure B-12.

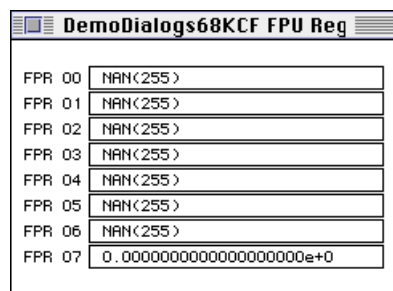
Figure B-12 Displaying the PowerPC floating-point registers

The PowerPC floating-point registers window displays 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 the Enter or Tab key. To undo a change, choose Undo from the Edit menu.

- The FPSCR value is displayed in binary format. Click on a bit to toggle its value.

Figure B-13 shows a sample 680x0 floating-point registers display.

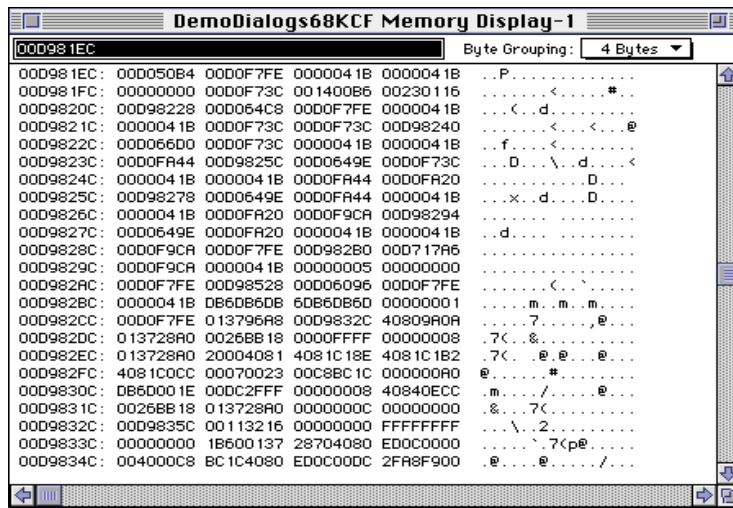
Figure B-13 Displaying the 680x0 floating-point registers

Debugger Menu Reference

Show Memory

Choose Show Memory from the Views menu (or press Command-M) to display memory. The debugger displays a scrollable memory display window that you can use to view memory in chunks of 256 MB (one segment). Figure B-14 shows a sample display.

Figure B-14 The memory display window



When you open a memory display window, the debugger tries to convert the current selection in the topmost window to a hexadecimal address. If it succeeds, it starts the memory dump from that address. Otherwise, the debugger starts the memory dump from the stack pointer.

To scroll to a particular address, use the scroll bar or enter the address in the box in the upper-left corner of the window and press Tab or Enter.

You can change memory 4 bytes, 2 bytes, or 1 byte at a time:

1. Click the value you want to change.

2. Edit the value.

If you change your mind, press the Escape key to stop editing without changing the value stored in memory.

3. Press the Enter or Tab key to make the change.

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

You can change the grouping used to display values in memory for the current session by selecting an alternate value from the Byte Grouping pop-up menu. Memory is grouped as 4-byte values by default. To change the default memory-grouping value, choose General Preferences from the Edit menu and select an alternative value.

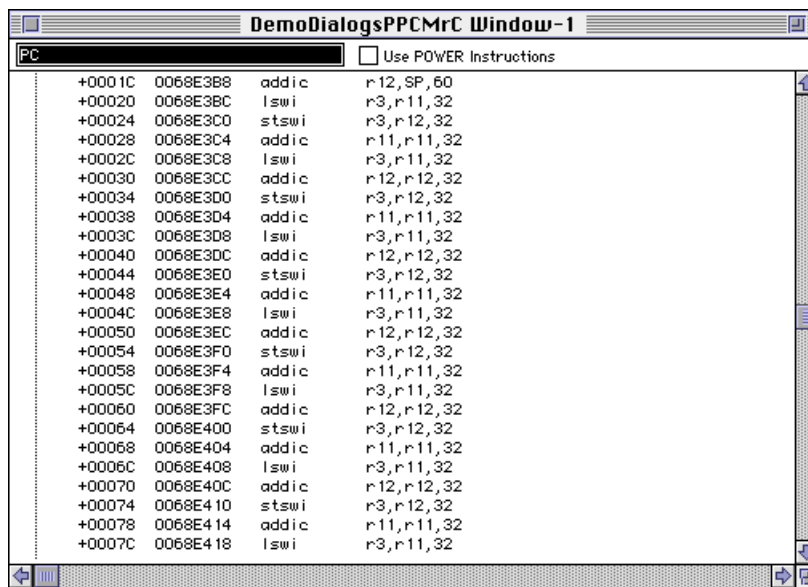
For additional information about memory organization on Power Macintosh computers, please see Chapter 5, “Low-Level Debugging.”

Debugger Menu Reference

Show Instructions

Not available for 68K Mac Debugger. Choose Show Instructions from the Views menu (or press Command-D) to display PowerPC instructions. The debugger displays a section of memory disassembled with the PowerPC disassembler in a window like the one shown in Figure B-15.

Figure B-15 The PowerPC instructions display window



When you choose Show Instructions, the debugger attempts to convert the selection in the topmost window to an address. If it succeeds, disassembly begins from that address. Otherwise, disassembly begins at the program counter. The disassembly address is shown in the box at the upper-left corner of the display. To change the address where disassembly begins, enter a new value in the box and press the Tab or Enter key to display the new disassembly.

You can view instructions using either the POWER instruction set mnemonics or the PowerPC mnemonics. The initial default setting is to display instructions using PowerPC mnemonics.

- To view instructions using POWER mnemonics, click the Use POWER Instructions checkbox. This affects the current session.
- To change the default setting to use POWER mnemonics, choose General Preferences from the Edit menu and click the appropriate checkbox.

You can set breakpoints in the instruction display window by placing the cursor in the breakpoint column and clicking to the left of the desired instruction. You can clear breakpoints or resume program execution in the usual way.

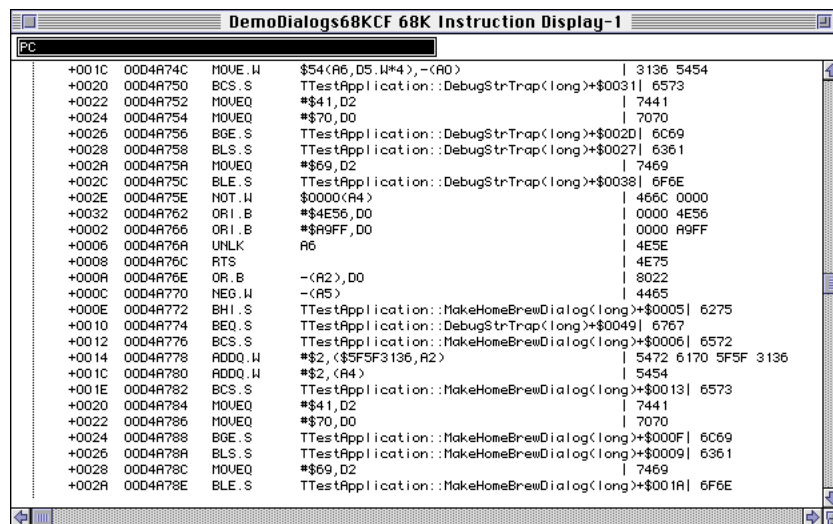
Debugger Menu Reference

For additional information about the instruction display, see Chapter 5, “Low-Level Debugging.”

Show 68K Instructions

Choose Show 68K Instructions (or press Command-8) to display a section of memory disassembled with the 680x0 disassembler. The debugger opens an instruction display window like the one shown in Figure B-16.

Figure B-16 The 680x0 instructions display window



When you choose Show 68K Instructions, the debugger attempts to convert the selection in the topmost window to an address. If it succeeds, disassembly begins from that address; otherwise, disassembly begins at the program counter. The disassembly address is shown in the box at the upper-left corner of the display. To change the address where disassembly begins, enter a new value in the box and press the Tab or Enter key to display the new disassembly.

Note

For Power Mac Debugger only, the 680x0 instruction display window is provided for reference only. It does not display the current PC. Moreover you cannot set breakpoints in this view. The validity of the disassembled instructions depends on your specifying a valid starting address for disassembly. ♦

Show Tasks

Choose Show Tasks to display the task browser, shown in Figure B-17. Use this option to manually target a running process. Select the process to target from those shown in the

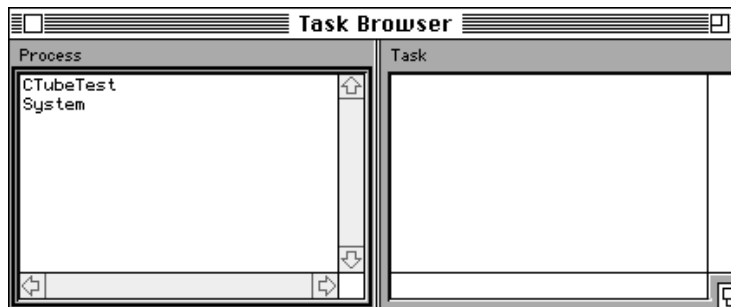
Debugger Menu Reference

Task Browser window. Then choose Target “*process name*” from the Control menu. One or two progress meters appear while the debugger reads the process information. Then a registers window opens for the newly targeted process. (The initial window contents are invalid because the process is not stopped.) To untarget a selected process, choose Untarget “*process name*” from the Control menu.

IMPORTANT

The Macintosh Debugger maps executable code to symbolic files by name. For 680x0 applications, the name of the symbolic file should be the name of the application with .NJ or .SYM appended. For example, the application Sample automatically maps to the symbolics file Sample.NJ (or Sample.SYM). For Power Macintosh applications, the name of the symbolic file should be the name of the application with .xcoff or .xSYM appended. For shared libraries, the name in the symbolics file should be the name in the 'cfrg' resource with .xcoff or .xSYM appended. If the application or shared library symbolics file does not have the same name, then the debugger is not able to automatically map the symbolics to the executable code, and you must use the Map Symbolics to Code command in the File Menu. ▲

Figure B-17 The Task Browser window



Show Fragment Info

Choose Show Fragment Info from the Views menu to bring up a window that displays a list of all fragments on the target machine, as shown in Figure B-18. The list includes all fragments on the target machine, not just those being debugged. Use the Refresh button to refetch all the data from the nub. The Refresh button is useful because this window is not automatically updated when fragments are created or deleted on the target machine.

Debugger Menu Reference

Figure B-18 The Fragment Info window

Fragment Info					
Process Name	Section Name	Address	Size	Dbg?	Type
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$00682750	\$000ADB1C	Yes	Code
DemoDialogsPPCMrC	DemoDialogsPPCMrC	\$007487E0	\$000287FA	Yes	Data
DemoDialogsPPCMrC	ObjectSupportLib	\$00886A40	\$00003E74	Yes	Code
DemoDialogsPPCMrC	ObjectSupportLib	\$00145B70	\$00000278	Yes	Data
DemoDialogsPPCMrC	StdCLib	\$00D76170	\$0000EBC4	Yes	Code
DemoDialogsPPCMrC	StdCLib	\$007460B0	\$00002710	Yes	Data
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$40A03B40	\$00022E2C	Yes	Code
DemoDialogsPPCMrC	InterfaceLib.409F3710	\$000153C0	\$00007384	Yes	Data
DemoDialogsPPCMrC	MathLib.40A33BA0	\$40A34AF0	\$000120D8	Yes	Code
DemoDialogsPPCMrC	MathLib.40A33BA0	\$00743910	\$00002778	Yes	Data
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$409EDEA0	\$00004438	Yes	Code
DemoDialogsPPCMrC	MixedMode.409ED7F0	\$00011D50	\$00000EAB	Yes	Data
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$40A2E900	\$00004C50	Yes	Code
DemoDialogsPPCMrC	PrivateInterfaceLib.40A27D90	\$0000F320	\$00002A08	Yes	Data
DemoDialogsPPCMrC	MixedMode.70B60*1	\$00071230	\$000044A8	Yes	Code
DemoDialogsPPCMrC	MixedMode.70B60*1	\$000E1260	\$00000EB0	Yes	Data
DemoDialogsPPCMrC	ProcessMgrSupport.606B0*1	\$00060920	\$00000AA3	Yes	Code
DemoDialogsPPCMrC	ProcessMgrSupport.606B0*1	\$000E2130	\$00000158	Yes	Data

The Show Exports button brings up a window that lists all the exports for the selected item, as shown in Figure B-19. (Show Exports is not available for 68K Mac Debugger.) You can sort the data in the Show Exports window in one of three ways: by symbol name, by address, or by symbol type. To sort the list, just click the column title to be used as the sort key. For example, click Symbol Name to sort by name.

If you double-click a row that displays a Code Vector symbol type, then the debugger opens an instruction window at the dereferenced address for the symbol. If the row displays a data symbol, then you can double-click it to open a memory window at the specified address.

Figure B-19 The Show Exports window

StdCLib		
Symbol Name	Address	Symbol Type
abort	\$00746CA4	Code Vector
abs	\$00746518	Code Vector
access	\$00746B54	Code Vector
asctime	\$00746524	Code Vector
atexit	\$00746FE0	Code Vector
atof	\$00746530	Code Vector
atoi	\$0074653C	Code Vector
atol	\$00746548	Code Vector
binhex	\$00746554	Code Vector
bsearch	\$00746560	Code Vector
calloc	\$0074656C	Code Vector
clearerr	\$00746998	Code Vector
clock	\$00746578	Code Vector
close	\$00746B60	Code Vector
ConvertTheString	\$00746830	Code Vector

Debugger Menu Reference

Find Code For...

Choose Find Code For from the Views menu (or press Command-F) to display a dialog box that you can use to specify the name of a routine that the debugger is to locate in your source code file. Figure B-20 shows the Find Code For dialog box.

Figure B-20 The Find Code For dialog box



Enter the name of the routine you want to find in the dialog box's text field box and click OK. It is also possible to bypass the dialog box by selecting the name of the routine you want to find and choosing the next item from the Views menu, Find Code For ""; this item is described next.

Find Code For ""

Select a routine name in the browser window and choose Find Code For "" (or press Command-F) to display source code for the specified routine in the browser. Once you select the routine name, the Find Code item in the Views menu is changed to read Find Code For plus the name of the selected routine. For example, if you select doClick, the menu item becomes Find Code For doClick.

Source File For ""

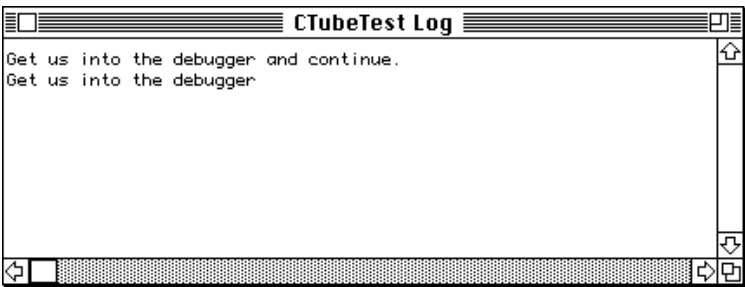
Select a routine name and choose Source File For "" from the Views menu to display the name of the source file containing the routine in the browser window. Once you select a routine name, it is added to the menu item. For example, if you select doClick, the menu item becomes Source File For doClick.

Show Log Window

Choose Show Log Window from the Views menu to display the log window. Figure B-21 shows an example of the log window.

Debugger Menu Reference

Figure B-21 The log window



The debugger creates a log window and writes the output of `DebugStr` calls to this window. You can save the contents of this window to an MPW text file by choosing **Save Log As** from the **File** menu.

Control Menu

You use commands in the **Control** menu to control the execution of your program, to display and clear breakpoints, and to show or hide the floating control palette. The **Control** menu is shown in Figure B-22. The following subsections describe the menu items in the order of their appearance.

Figure B-22 The Control menu



Debugger Menu Reference

You can also execute most of the items listed in the Control menu by choosing the corresponding item from the floating control palette. The correspondence between Control menu items and palette icons is shown in Table B-2.

Table B-2 Control menu items and floating control palette icons

Menu item	Run	Stop	Step	Step Into	Step Out	Animate
Palette icon						

Run “”

Choose Run from the Control menu (or press Command-R) to resume execution of the target application.

Stop “”

Choose Stop from the Control menu to halt execution of the target application. Once the target application is stopped, you can set breakpoints and resume controlled execution of the target. The Stop menu item is supported only when using Power Mac Debugger’s low-level nub.

Kill “”

The Kill command is implemented only for source-level debugging nubs—68K Mac DebugServices and Power Mac DebugServices.

Resume “”

Choose Resume from the Control menu to resume execution of a thread or task that had been marked as ineligible for execution by the debugger. To change the state of a thread or task, select it in the Task Browser window (displayed when you choose Show Tasks from the Views menu), and then select Resume to place the thread or task back into the ready state.

Propagate Exception

Choose Propagate Exception from the Control menu (or press Command-Option-R) to propagate an exception to your application’s exception handler. Choose this item following an exception if your application includes an appropriate exception handler or if you want the exception passed to the System Error Handler.

Debugger Menu Reference

In some cases you might get memory access errors that were not generated by your application. If you determine that these errors originated elsewhere, you should choose Propagate Exception to propagate these errors to the Memory Manager's error handler.

Target ""

Choose Target "*Process Name*" from the Control menu to manually target a running process previously selected through the Show Tasks option of the Views menu. You should see one or two progress meters while the debugger reads the process information. Then a registers window is opened for the newly targeted process. The initial contents of the registers window are not valid since the process is not stopped.

Untarget ""

Choose Untarget "*Process Name*" from the Control menu to untarget a previously targeted process.

Launch

Choose Launch from the Control menu to launch an application and have it stop before executing `main`. This command is helpful for debugging initialization code. The Launch command works only for one-machine debugging; to break on launch using two-machine mode, hold down the Control key on the remote machine as you double-click the application.

Step

Choose Step from the Control menu (or press Command-S) to step through the source code, stepping over procedure calls. Stepping takes place at the source level if a non-assembly code view is the front window and at assembly level otherwise. If the frontmost window is neither a source-level nor an assembly-level view, stepping takes place in the last mode used. Default stepping is at the assembly level. Thus if you step for the first time and any window other than a source-level view is active, an assembly-level step is taken.

If a browser window is the front window and you step into code that is in another browser, that window is brought to the front. If you step into code that has no associated source code, the debugger brings up an instruction window to display the program counter.

Step Into

Choose Step Into from the Control menu (or press Command-I) to step into a procedure call. Choose Step to step over procedure calls.

Debugger Menu Reference

Step Out

Choose Step Out from the Control menu (or press Command-T) to step out of the current (called) routine and to stop at the next instruction of the calling routine.

Step To Branch

Choose Step To Branch from the Control menu (or press Command-B) to cause the debugger to automatically step over all instructions until it reaches any branch instruction, at which point it stops. This command can be used at source or assembly level, but makes most sense when used at assembly level.

Step To Branch Taken

Step To Branch Taken has the same effect as Step To Branch, except that it stops only if the branch is actually going to be taken. This command can be used at source or assembly level, but makes most sense when used at assembly level.

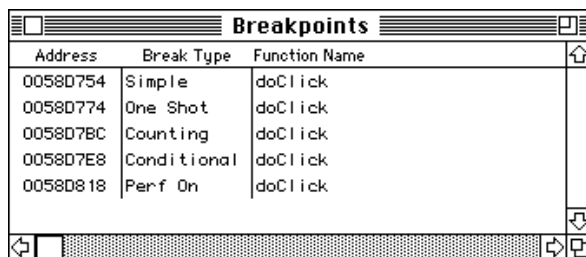
Animate

Choose Animate from the Control menu to continuously execute the next Step, Step Into, or Step Out command that you choose. The target application stops when it reaches a breakpoint. You can select Animate again to disable this mode even while the target is executing. The menu item is checked when the animate mode is enabled.

Show Breakpoints List

Choose this item to display the currently set breakpoints. The debugger displays a window similar to the one shown in Figure B-23.

Figure B-23 The breakpoints window



Address	Break Type	Function Name
0058D754	Simple	doClick
0058D774	One Shot	doClick
0058D7BC	Counting	doClick
0058D7E8	Conditional	doClick
0058D818	Perf On	doClick

The window shows the function name, the function's starting address, and the type of breakpoint set at that address.

Debugger Menu Reference

- If you double-click the function name, the source view of the function is displayed in the browser window.

If the function name is shown as “???”, the debugger cannot display code for that function. In this case, the debugger cannot find symbolic information or embedded names mapped to the address where the break has been set.

- If you double-click the address, the disassembly for the function is shown in the instruction window.

If the address column is blank for a breakpoint, the breakpoint has not actually been set on the target machine yet. This happens if you set a breakpoint in the browser window when the target process is not stopped or if you open a symbolic information file before launching the target application.

If you select a breakpoint in the breakpoints window and press Delete, the breakpoint is removed from your application and from the window. For more information about setting breakpoints, see “Setting and Clearing Breakpoints” on page 3-9.

Clear All Breakpoints

Choose Clear All Breakpoints from the Control menu to clear all breakpoints.

Show/Hide Control Palette

Use this command to open or close the floating control palette. When the floating control palette is open, this command becomes Hide Control Palette. When the floating control palette is closed, this command becomes Show Control Palette.

You can also close the palette by clicking its close box.

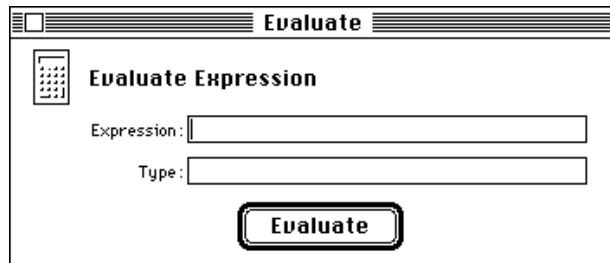
Evaluate Menu

You use items in the Evaluate menu to display the value of an expression, a symbolic name or address, or “this” in C++. The “View as” commands, included in the lower part of the menu, allow you to display values in a variety of formats. The following subsections describe Evaluate menu items in the order of their appearance. The Evaluate menu is shown in Figure B-24.

Figure B-24 The Evaluate menu

Show Evaluation Window

When you choose Show Evaluation Window from the Evaluate menu, the debugger displays a dialog box like the one shown in Figure B-25.

Figure B-25 The Evaluate dialog box

Enter the variable name, address, or expression in the Expression text field.

- If you specify an address, you must also enter a string in the Type text field that specifies the data type used to interpret the contents in memory; for example: "short" or "windowrecord*".
 - If you specify a name, the Type entry is optional.
 - If you specify an expression, the Type entry is optional.
- For additional information about the evaluation of expressions, see Appendix A.

Evaluate ""

Choose Evaluate from the Evaluate menu (or press Command-E) to display the value of a variable or expression selected in any code view. The text of the menu item is completed with the name of your selection. For example, if you select the variable

Debugger Menu Reference

`MyCounters`, the command becomes `Evaluate MyCounters`. For complete information about the grammar of expressions, see Appendix A.

Evaluate “this”

If you select `Evaluate “this”` from the Evaluate menu and the target program is stopped in a C++ method, the debugger displays the value of the instance of the current class.

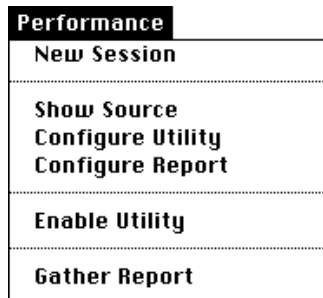
Displaying Values in Different Formats

To change the display format of a selected variable in any code view, choose one of the following items from the Evaluate menu:

Item	Effect
View as Default Type	Display the selected value using the appropriate default format. The default format for numbers is decimal; for unsigned longs, the default type depends on the current preference settings (set in dialog box you get by choosing General Preferences from the Edit menu); the default format for strings is an array of chars.
View as Character	Display the selected value as a character.
View as Decimal	Display the selected value as a 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 Str255	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.

Performance Menu

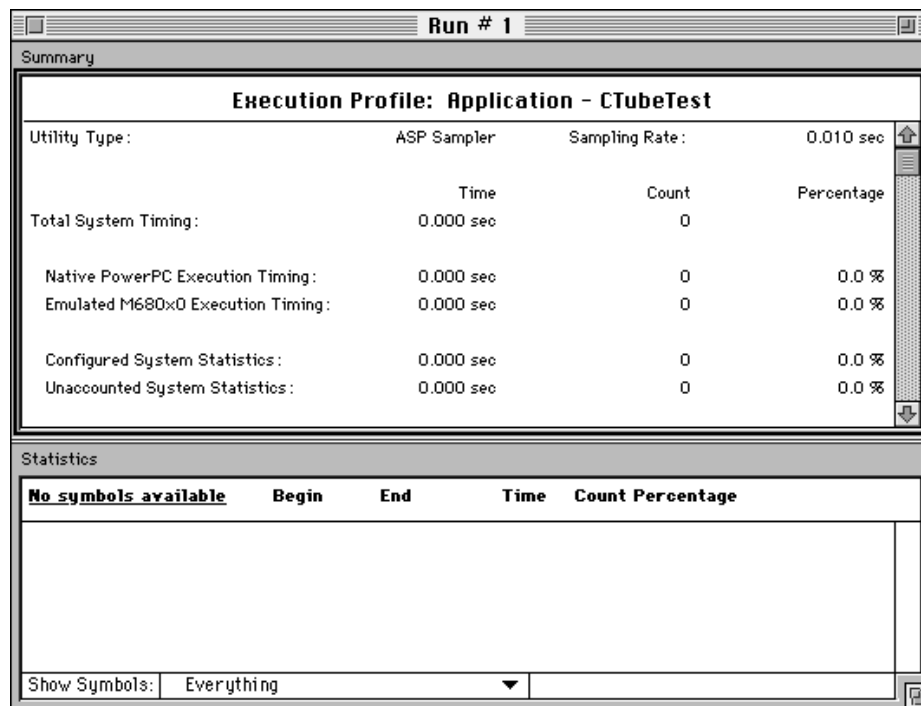
Not available for 68K Mac Debugger. You use items in the Performance menu to operate the performance analysis tools that have been integrated with the debugger. The Performance menu is shown in Figure B-26. The following subsections describe Performance menu items in the order of their appearance. For additional information about using the Adaptive Sampling Profiler, see Chapter 4.

Figure B-26 The Performance menu

New Session

Choose New Session from the Performance menu to initiate a new sampling session and to open a performance document window without any data, like the one shown in Figure B-27.

The window's title identifies the performance session. In Figure B-27 it is Run #1. If you were to end this session and start another for the same application, the ASP would name the window for the second session Run #2. The automatic numbering lets you distinguish sampling sessions for the same application.

Figure B-27 A blank performance document window

Debugger Menu Reference

The name of the application to be measured appears just below the window title. The application's name is the same as the name of the fragment that contains the `main` function for the current process.

The performance document window contains two scrollable views: the *summary view* and the *statistics view*. When you first open the performance document, before you have collected performance data and generated a report, these views do not have any data.

- The summary view displays information about the total time the application has executed, the total number of sample hits, the sampling rate used for this invocation, and the number of hits for routines in shared libraries.
- The statistics view provides more detailed profiling information for each fragment sampled and, in the case of fragments for which symbolic information files exist, for each routine in the fragment.

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.

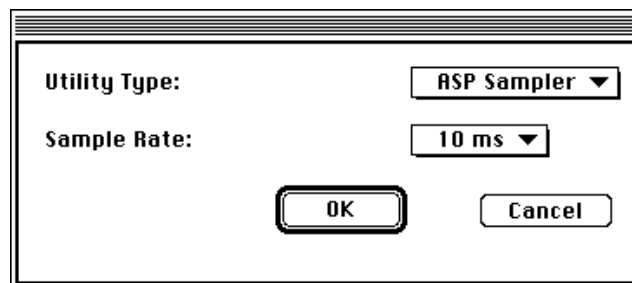
Show Source

Select a function name in the statistics view and choose Show Source to display the source code in the browser. You can also double-click the function's name in the statistics view to display the source code.

Configure Utility

Choose Configure Utility from the Performance menu to select a sampling rate for the Adaptive Sampling Profiler; the debugger displays a dialog box like the one shown in Figure B-28.

Figure B-28 The utility configuration dialog box



The default sampling rate is initially set at 10 ms. This means that every 10 ms the ASP issues an interrupt and records the current PC value. To change the sampling rate, choose another value from the pop-up menu.

Debugger Menu Reference

Take care in selecting a sampling rate. A time interval that is too large can result in a partial view of your program's performance, whereas a time interval that is too small can result in excessive interrupt processing that tends to perturb the execution of the system. There is no hard and fast formula for choosing a sampling rate. Instead, you should try multiple sessions with different sampling rates to arrive at the best overall picture of your application's performance.

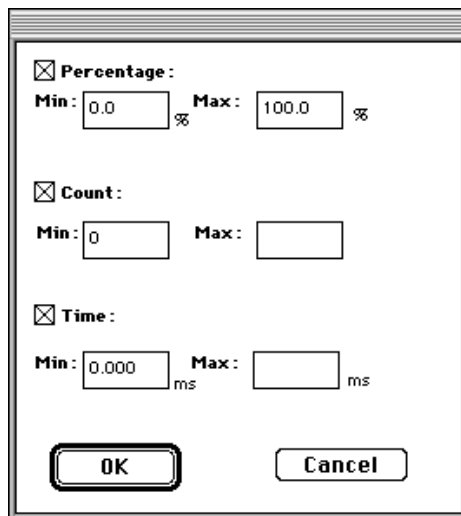
Unless you have a specific reason to change the sampling rate value, you should begin by working with the default setting. For additional information about sampling rate values, see "Evaluating Performance Data" on page 4-15.

Once you have chosen a sampling rate and resumed execution, it is not possible to reconfigure the sampling rate.

Configure Report

You use Configure Report from the Performance menu to have the Adaptive Sampling Profiler filter out information about selected modules or performance data from its final report. When you choose this item, the debugger displays a report configuration dialog box like the one shown in Figure B-29.

Figure B-29 The report configuration dialog box



This dialog box consists of statistics filters. You supply minimum and maximum values for one or more of these in order to filter out information based on performance criteria.

1. Click one of the checkboxes labeled **Percentage**, **Count**, or **Time** on the left side of the dialog box.

Leaving a checkbox blank means that this filter is inactive.

2. Specify a minimum and maximum value in the text fields labeled **Min** and **Max**.

Debugger Menu Reference

For example, if you select Percentage and specify 1% as a minimum value and 80% as a maximum value, the ASP will not display information for routines that have received less than 1% or more than 80% of the hits gathered. Leaving the Min or Max text field blank means that the field has no bounds.

It is possible to 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, the ASP will filter out information according to whichever of these values is the final minimum.

Enable/Disable Utility

When the ASP is disabled, this item reads Enable Utility; when the ASP is enabled, this item reads Disable Utility. You use this command to manually enable and disable performance measurement during controlled program execution in order to measure selected routines in your application.

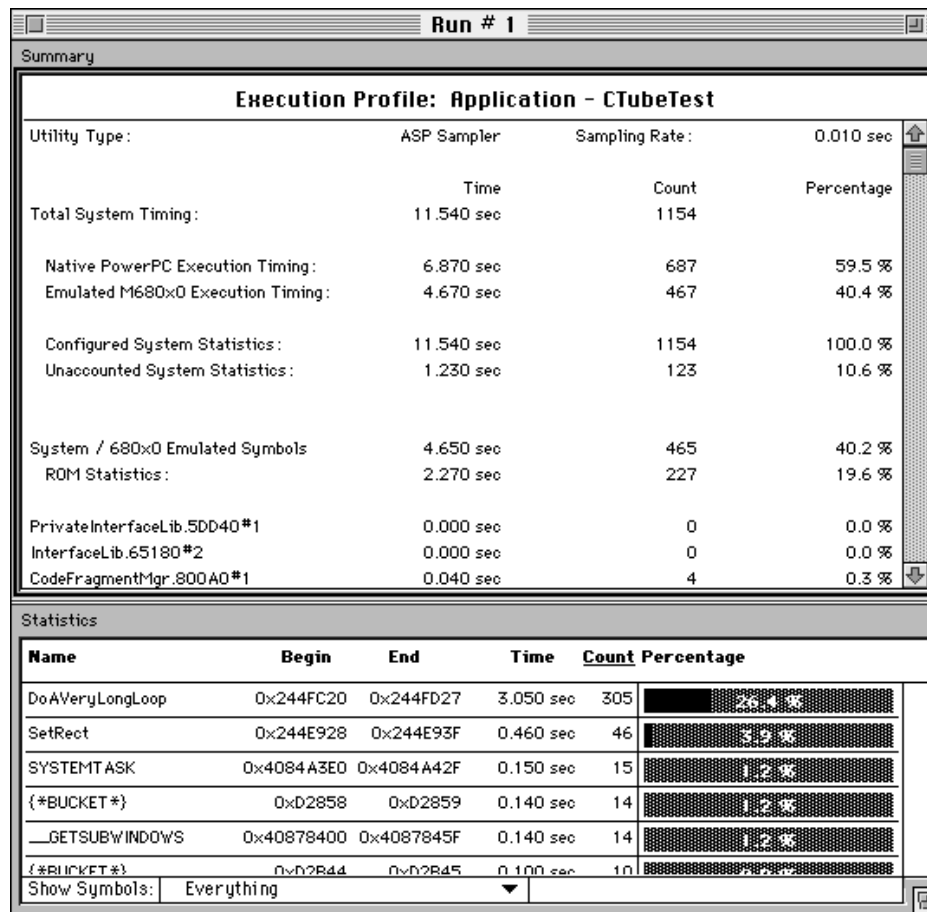
You can enable or disable the sampler whenever your program reaches a breakpoint. After enabling performance measurement, you can choose Run or Step from the Control menu to resume execution of your application; the ASP can collect data in either state.

You can also use the breakpoint dialog box to set performance breakpoints in your application. For additional information, see “Measuring Selected Routines” on page 4-8.

Gather Report

Choose Gather Report from the Performance menu to generate a performance report after the ASP has collected performance data for your program. Before you choose this item, you must stop the process running on the target machine. When the ASP has collected and sorted information for the code being measured, it updates the performance document window so that it looks like the one shown in Figure B-30.

Debugger Menu Reference

Figure B-30 A performance document window after a report has been gathered

The following sections explain the information displayed in this window. For additional information about how the ASP identifies hits with distinct memory regions, see “How the ASP Presents Collected Data” on page 4-9.

The Summary View

The top line of the summary view identifies the utility that produced the current report and how that utility was configured. In this case, the utility is the ASP Sampler, configured to sample the PC every 10 ms.

The next grouping of information shown in the summary view displays data in five categories: Total System Timing, Native versus Emulated Execution Timing, Configured System Statistics versus Unaccounted Statistics, System/680x0 Emulated Symbols, and ROM Statistics.

- Total System Timing represents the total number of hits recorded and the total time for those hits. The lines indented under Total System Timing represent different types of breakdowns for the total hits.

Debugger Menu Reference

- **Native PowerPC Processor Execution Timing and Emulated M680x0 Execution Timing** present statistics that further break down the Total System Timing data according to the native or emulated PCs sampled.
- **Configured System Statistics** represents the total number of hits that the ASP was able to map to a specific fragment. It also represents the difference between Total System Timing and Unaccounted System Statistics, which represents the total number of hits accumulated either as a result of ASP registration errors or as a result of filtering criteria.
 ASP registration errors occur when the address range of a hit overlaps the boundary of an executable fragment. In such cases, the ASP cannot definitively map a hit to a specific fragment and assigns such hits to Unaccounted System Statistics.
 If you use the report configuration dialog box (shown in Figure 4-9 on page 4-14) to exclude information from the final report, hits accumulated for the excluded regions are included in Unaccounted System Statistics.
- **System/680x0 Emulated Symbols** represents the number of hits collected for code executing outside of any fragment associated with the application being measured. Most applications spend a good deal of time in this region, especially applications that make heavy use of the user interface. The address ranges associated with this section include the ROM.
- **ROM Statistics** represents the total number of hits that were associated with a ROM address range. If there are system symbols or MacsBug symbols associated with a ROM address range, more detailed information for these routines is shown in the statistics view.

Following these summaries is a summary breakdown of statistics per fragment. Each line displays summary information for one fragment. These are listed in ascending order by the beginning address of each fragment.

The Statistics View

The statistics view displays information for all fragments called by your application and for all routines in those fragments if a symbolic information file for the fragment has been loaded into the debugger. This default setting is signaled by the label Everything in the Show Symbols pop-up menu in the lower-left corner of the performance document. Also by default, the statistics displayed for the modules and fragments in the statistics view are listed in descending order according to the number of hits registered for each fragment.

- To display data for a specific fragment, choose the name of the fragment from the Show Symbols pop-up menu.
- To change the sort order for the data displayed, click one of the column headings Name, Begin, End, Time, Count, or Percentage. The heading you select is underlined.
 - ☐ Name sorts the data in alphabetical order by module and fragment name.
 - ☐ Begin sorts the data in ascending order by address: statistics for the module or fragment at the lowest address are shown first.
 - ☐ Time, Count, and Percentage sort data in descending order, with the most frequently used shown first.

Debugger Menu Reference

In certain cases the ASP is not able to attribute a name to a memory region where a hit was registered. For additional information, see “How the ASP Presents Collected Data” on page 4-9.

Extras Menu

You use items in the Extras menu to enter MacsBug, show the current PC, execute a debugger extension, enable/disable user breaks, search memory for a specific value, or save a snapshot of the frontmost window. The Extras menu is shown in Figure B-31. The following subsections describe the menu items in the order of their appearance.

Figure B-31 The Extras menu



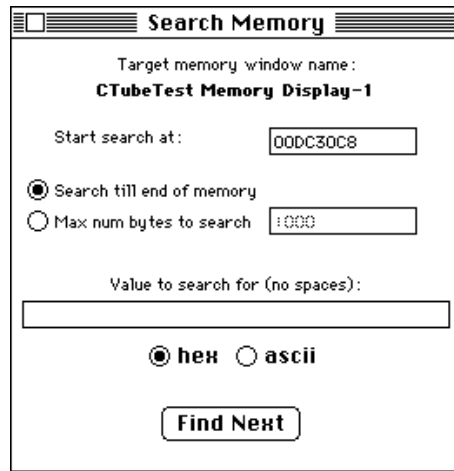
Show PC

Choose Show PC from the Extras menu (or press Command-/) to display the source code for the address currently stored in the program counter register.

Search Memory

Choose Search Memory from the Extras menu to display the Search Memory window, shown in Figure B-32. The Search Memory menu item is enabled only when a memory window is the frontmost active window. The debugger supports searching for ASCII or hexadecimal values. The Search Memory window is tied to the specific instantiation of a memory window and closes when that window closes.

Debugger Menu Reference

Figure B-32 The Search Memory window

Enter MacsBug

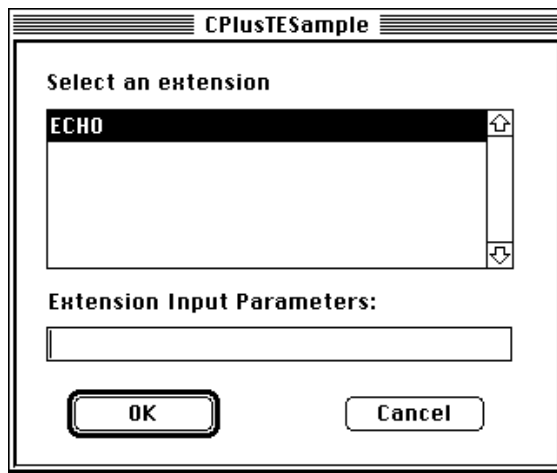
Choose Enter MacsBug from the Extras menu to invoke MacsBug on the target machine. You can use MacsBug to examine the execution of emulated code on your Power Macintosh computer. For information about MacsBug commands, see the *MacsBug Reference and Debugging Guide*.

Stop For DebugStrs

Choose Stop For DebugStrs from the Extras menu to enable user breaks. If user breaks are enabled, the menu command reads Don't Stop For DebugStrs; choose this item to disable user breaks.

Execute Debugger Extension...

Choose Execute Debugger Extension from the Extras menu to display available debugger extensions. The debugger displays a dialog box like the one shown in Figure B-33.

Figure B-33 Listing available debugger extensions

Select a debugger extension by clicking its name in the scrollable window. If the extension requires that you specify one or more parameters, enter these in the text field labeled Extension Input Parameters. Use spaces to separate multiple parameters. To execute the extension, click OK (or press the Return key.)

For more information about writing, building, and installing debugger extensions, see Chapter 6, "Debugger Extensions."

Snapshot Active Window

Choose Snapshot Active Window from the Extras menu to save a snapshot of the current frontmost window. When you expect to change the state of the frontmost window, you can save a snapshot to use for comparison later.

Windows Menu

You use items in the Windows menu to activate one of the windows that are already opened on the desktop. The currently active window is designated by a checkmark.

Debugger Shortcuts

Contents

General	C-3
Browser Window	C-3
Breakpoints Window	C-3
Stack Window	C-4

Debugger Shortcuts

This appendix lists some shortcuts you can use with the Macintosh Debugger. Unless otherwise specified, these shortcuts apply to both 68K Mac Debugger and Power Mac Debugger.

General

- To execute a break-on-launch function, hold down the Control key when launching the target application. You enter the debugger as soon as the application is launched.
- To create a memory display window starting at a certain address, select that hexadecimal address and press Command-M. This command also changes the address at the top of the memory display window and refreshes the window. To create a new memory display window instead, press Shift-Command-M.
- To create a Power Mac Debugger instruction display window starting at a certain address, select that hexadecimal address and press Command-D.
- To create a Macintosh Debugger instruction display window starting at a certain address, select that hexadecimal address and press Command-8.
- To change the program counter, click the program counter arrow and drag it.

Browser Window

- To create a code view window, hold down the Option key; then click anywhere in the code view of the browser window and drag.
- To set a one-shot breakpoint, press Command-Option-click in the breakpoint column of a browser window.
- To display a breakpoint dialog box (from which you can select the type of breakpoint you want), press Option-click in the breakpoint column.

Breakpoints Window

- To delete a breakpoint, select the breakpoint in the breakpoints window and press Delete. The debugger removes the breakpoint from your application and from the breakpoints window.

Stack Window

- To create a copy of the values pane for a specific variable, click anywhere in the values pane of the stack window; then hold down the Option key and drag. You must first have selected a variable in the upper-left pane.
- To display an instruction window with disassembly beginning at a certain PC address, double-click that PC address.
- To display a memory window with the memory dump beginning at a certain frame address, double-click that frame address.
- To display a browser window or an instruction window, double-click a function name.

Creating Custom Unmangle Schemes

Contents

Unmangle Schemes for 680x0 Code	D-3
Unmangle Schemes for PowerPC Code	D-5

Creating Custom Unmangle Schemes

The Macintosh Debugger is able to load any unmangle scheme as a code resource and use its algorithm to unmangle symbols within the symbolic file. This capability lets the user use multiple compilers with varying unmangle schemes with the Macintosh Debugger. This appendix outlines the steps necessary to create your own unmangle stand-alone code resource for use by the Macintosh Debugger with either 680x0 or PowerPC code.

Unmangle Schemes for 680x0 Code

1. Make sure that the function header and parameter list for your unmangle scheme look like the following:

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

2. If global variables are to be used, make sure that an A5 world is set up for their use. Create a wrapper around your code as follows:

```
A5RefType A5Ref;
long oldA5;
long theReturnValue = 1;

MakeA5World(&A5Ref);
oldA5 = SetA5World(A5Ref);
.
.
.
SetA5(oldA5);
DisposeA5World(A5Ref);
```

The above routines are defined in `SAGlobals.c`; be sure to include them in your link. They look like this:

```
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) {
```

Creating Custom Unmangle Schemes

```

        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);
}

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 and 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);

```

3. The makefile for your 680x0 stand-alone code resource looks like the following:

```

#   File:          unmangle.make
#   Target:        unmangle
#   Sources:       SAGlobals.c
#                  unmangle.c
#   Created:       Tuesday, September 6, 1994 03:54:00 PM

OBJECTS =  
    unmangle.c.o  
    SAGlobals.c.o

unmangle ff unmangle.make {OBJECTS}
Link -rt CUST=1000 -t CODE -c '????' -m unmangle -sg unmangle  
    {OBJECTS}  
    "{CLibraries}"StdClib.o  
    "{Libraries}"Runtime.o  
    -o unmangle
SAGlobals.c.o f unmangle.make SAGlobals.c
    C -r SAGlobals.c
unmangle.c.o f unmangle.make unmangle.c
    C -r unmangle.c

```

Creating Custom Unmangle Schemes

4. Place the created resource in the Macintosh Debugger Preferences folder. After launching the debugger, select your scheme as the chosen unmangle scheme in the General Preferences command of the Edit menu.

This code resource can be read by 68K Mac Debugger and Power Mac Debugger, running on any combination of machines.

Unmangle Schemes for PowerPC Code

1. Make sure that the function header and parameter list for your unmangle scheme looks like the following:

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

2. The makefile for a PowerPC code resource is a little more complex than that for the 680x0 code resource. Here is an example:

```
# VARIABLE DEFINITIONS
SrcName      = unmangle
ResourceName= {SrcName}
Objs         = {SrcName}.c.o                                @
              "{PPCLibraries}"InterfaceLib.xcoff           @
              "{PPCLibraries}"StdCLib.xcoff
LIBEQUATES   = -l InterfaceLib.xcoff=InterfaceLib          @
              -l StdCLib.xcoff=StdCLib

# DEPENDENCIES
.c.o         f .c
PPCC -appleext on {default}.c -o {default}.c.o
{ResourceName} ff {Objs} {SrcName}.r1
{SrcName}.r2 {ResourceName}.make
PPCLink {Objs} -o {SrcName}.xcoff -main unmangle
MakePEF {SrcName}.xcoff -b -o {SrcName}.pef {LIBEQUATES}
Rez {SrcName}.r1 -o {SrcName}.pef -c RSED -t 'CODE'
Rez {SrcName}.r2 -a -m -o {SrcName} -c RSED -t 'CODE'
Setfile {SrcName} -a B
```

The compile and link are fairly straightforward. Here's how the Rez commands break down:

- a) unmangle.r1 looks like this:

```
Read 'CUST' (1000) "unmangle.pef";
```

This defines the code resources ID and type.

Creating Custom Unmangle Schemes

b) unmangle.r2 looks like this:

```
#include "MixedMode.r"
type 'CUST' as 'rdes';
resource 'CUST' (1000)
{
    $00000FF1, /* this is the Mixed Mode Manager procInfo value */
    $$Resource ("unmangle.pef", 'CUST', 1000)
};
```

This gives the code fragment a routine descriptor, which in turn gives the debugger directions to the new unmangle scheme.

- 3. Place the created resource in the Macintosh Debugger Preferences folder. After launching the debugger, select your scheme as the chosen unmangle scheme through the General Preferences command of the Edit menu.**

This code resource can be read only by the Power Macintosh–hosted Power Mac Debugger.

Targeting Code Resources by Resource Type

Contents

About ResourceTracker	E-3
Using ResourceTracker	E-3

Targeting Code Resources by Resource Type

This appendix describes ResourceTracker, an application that applies only to 68K Mac Debugger.

About ResourceTracker

ResourceTracker is an application that allows the user to target code resources of a resource type other than the standard 'CODE', such as 'INIT' or 'CDEV'. The targeted resource list is kept in the 68K DebugServices Prefs file. ResourceTracker writes the resource types to be debugged to this file, and the 68K Mac DebugServices nub reads the targeted resource list from the file.

Using ResourceTracker

Here are the steps you follow to target and stop targeting a resource:

1. To target another code resource type, launch ResourceTracker.

ResourceTracker opens the 68K DebugServices Prefs file and reads the current list of targeted resource types for the 68K Mac DebugServices nub. ResourceTracker then displays this list in the Resource Tracker window.

2. To add a resource type, select the desired resource type from the list, enter its name in the text edit box, and then click Add.

3. To stop targeting a resource, select that resource and then click Delete.

These changes are written to the 68K DebugServices Prefs file when you quit ResourceTracker. The standard code resource type 'CODE' is tracked automatically, and you cannot delete it from the tracking list.

4. After you add or delete a resource, you must quit the ResourceTracker application and relaunch 68K Mac Debugger to make the change effective. You also need to reboot the machine being debugged.

Glossary

680x0 Any member of the Motorola 68000 family of microprocessors. In the illustrations in this book, *680x0* refers to any Macintosh Quadra (or higher) computer.

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

680x0-based Macintosh computer Any computer containing a 680x0 central processing unit that runs Macintosh system software. See also **Power Macintosh computer**.

68K Mac Debugger The part of the Macintosh Debugger that provides source-level debugging of 680x0 code running on either a 680x0-based or Power Macintosh computer.

ASP (Adaptive Sampling Profiler) A sampling utility that allows you to measure how often sections of code execute.

browser window A three-pane window displayed when you open a symbolic information file. You can use this window to view the source code for selected functions in your target program. This and the control palette are the debugger's two main ways of controlling execution of your program.

bucket In performance measurement, a counter associated with a discrete memory range.

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

code fragment See **fragment**.

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

code view The lower pane of the debugger browser window, used to display code at the source or assembly level.

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

control palette A floating window that displays information about the target process and its current status. This and the browser window are the debugger's two main ways of controlling execution of your program.

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

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

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. This time does not include the time spent in any routine called by that routine.

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

FPSCR See **Floating-Point Status and Control Register**.

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

hit In performance measurement, to increment a counter when the sampled PC address falls within the memory range associated with that counter.

host In debugging on a two-machine setup, the part of the debugger that presents information transmitted by the nub via a user interface. Compare **target**.

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

Macintosh Debugger A tool for debugging applications written for Power Macintosh and 680x0-based Macintosh computers. See also **Power Mac Debugger** and **68K Mac Debugger**.

measured overhead The portion of the total overhead that is contained in the sample timings. See also **total overhead**.

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

'ndcd' resource The type of resource used to define or customize debugger functions.

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

nub In debugging on a two-machine setup, the part of the debugger that resides on the target machine and provides information about the state of the target machine to the host via a serial or network cable.

one-shot breakpoint A breakpoint that allows you to set a breakpoint and resume execution of the target application with a single mouse click.

PEF See **Preferred Executable Format**.

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

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

Power Mac Debugger The part of the Macintosh Debugger that provides (1) source-level debugging of PowerPC code and (2) low-level debugging of PowerPC, emulated 680x0, and assembly-language code.

Power Macintosh computer Any computer containing a PowerPC central processing unit that runs Macintosh system software. See also **680x0-based Macintosh 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 The format of executable files used for PowerPC applications and other software running on Macintosh computers. See also **Extended Common Object File Format**.

program counter (PC) A register in the CPU that contains a pointer to the memory location of the next instruction to be executed.

prolog A standard piece of code at the beginning of a routine that sets up the routine's stack frame and saves any nonvolatile registers used by the routine.

Red Zone The area of memory immediately above the address pointed to by the stack pointer. The Red Zone is reserved for temporary use by a function's prolog and as an area to store a leaf routine's nonvolatile registers.

RTOC See **Table of Contents Register**.

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

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

stack frame The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

statistics view The part of the performance document window that contains profiling information for a fragment's routines.

summary view The part of the performance document window that contains summary profiling information for a performance session.

switch frame A stack frame, created by the Mixed Mode Manager during a mode switch, that contains information about the routine to be executed, the state of various registers, and the address of the previous frame.

symbol A name for a discrete element of code or data in a fragment.

symbolic information file A file built by the MakeSym tool that contains a symbolic map which allows the debugger to map instructions back to a fragment's source code.

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 Macintosh Debugger.

TOC See **table of contents**.

total overhead The difference in the elapsed times when running your application with the ASP and running your application without it. See also **measured overhead**.

Index

Numerals

680x0 floating-point registers 3-27, B-16
680x0 general-purpose registers 3-26, B-15
680x0 registers, names of in expressions A-4
68K DebugServicesINIT extension 1-6
68K Mac Debugger
 components 1-6
 installing 2-5
 launching 2-6
68K Mac Debugger Prefs file 1-6
68K Mac DebugServices application 1-6

A

Adaptive Sampling Profiler. *See* ASP
addresses
 displaying value of 3-28
 evaluating B-28
address space 5-3
aligned data 5-4
Animate command 3-17, B-26
ASCII values, and searching 3-20
ASP 4-3 to 4-20
 allocating nodes 4-16
 collecting performance data 4-8
 data collection by 4-15
 disabling B-33
 enabling 4-6, B-33
 evaluating performance data 4-15
 flat-time measurement 4-4
 freeing nodes 4-19
 loss of information 4-18
 node allocation 4-16
 node tables 4-16
 overhead 4-19 to 4-20
 performance breakpoints 3-13, 4-8
 performance document
 creating 4-6, B-30
 printing 4-14
 saving 4-14
 views in 4-6, 4-11 to 4-13, B-31
 performance of shared libraries 4-4
 performance of stand-alone code 4-4
 performance report 4-9 to 4-13
 editing 4-13, B-32
 generating 4-9

 statistics view 4-11
 registration errors 4-19
 resonance effect 4-15
 restricting measurement 4-9
 sampling rate, specifying 4-7
 sorting statistics display 4-12, B-35
 splitting buckets 4-17
 summary 4-3
assertions, use of 3-15

B

break on launch 2-11, 4-5, C-3
breakpoint icons 3-9
breakpoints 3-9 to 3-15
 clearing 3-10, B-27
 conditional 3-13
 counting 3-12
 deleting B-27, C-3
 displaying 3-14, B-26
 one shot 3-11
 performance 3-13, 4-8
 setting 3-10, 3-14, 3-22, B-18
 simple 3-10
 types of 3-9
 user breaks 3-14
breakpoints list 3-14, B-26
browser window
 closing B-4
 displaying multiple code views 3-6
 parts of 1-9, 3-4
 printing code view 3-8
 program counter icon 3-17
 sample 3-5
 selecting a function in 3-5
 shortcuts for C-3
 updating PC in 2-13, B-7
buckets
 defined 4-3
 splitting 4-17
build process 2-3 to 2-4

C

C++ expression evaluation. *See* expressions

- C++ unmangle schemes 2-14, B-8
- callback routines 6-11 to 6-15
- calling conventions, stack-based 5-12
- C expression evaluation. *See* expressions
- Clear All Breakpoints command 3-10, B-27
- Clear command B-6
- Clipboard
 - clearing B-6
 - displaying contents of B-6
- Close command B-4
- closing a window B-4
- code profiling. *See* ASP
- code resources, targeting E-3
- code views 3-5
- CommandEntry function 6-4
- conditional breakpoints 3-13
- Condition Register 5-8, A-4
- Configure Report command B-32
- Configure Utility command B-31
- constants, in debugger extensions 6-9 to 6-10
- controlling execution 3-15
- Control menu 3-15, B-23 to B-27
 - Animate 3-17, B-26
 - Clear All Breakpoints 3-10, B-27
 - Hide Control Palette B-27
 - Kill "" B-24
 - Launch 3-18, B-25
 - Propagate Exception 3-16, B-24
 - Resume "" B-24
 - Run "" 3-16, B-24
 - Show Breakpoints List B-26
 - Show Control Palette 3-15, B-27
 - Step 3-16, B-25
 - Step Into 3-17, B-25
 - Step Out 3-17, B-26
 - Step To Branch B-26
 - Step to Branch 3-18
 - Step To Branch Taken B-26
 - Step to Branch Taken 3-18
 - Stop "" 3-16, B-24
 - Target "" 3-18, B-25
 - Untarget "" 3-18, B-25
- control palette 1-10, 3-15 to 3-17
 - and Control menu B-23 to B-27
 - display on launch 2-13, B-7
 - icons for 3-16
 - opening and closing B-27
 - use of 3-3
- Copy command B-5
- counting breakpoints 3-12
- Count Register 5-9, A-4
- Cut command B-5

D

- data alignment 5-4
- data storage 5-3
- data structures, in debugger extensions 6-10 to 6-11
- dcmdDrawLine function 6-13
- dcmdDrawText function 6-14
- dcmdGetNextChar function 6-12
- dcmdGetNextExpression function 6-13
- dcmdGetNextParameter function 6-12
- dcmdGetPosition function 6-11
- dcmdGetTrapName function 6-14
- dcmd.h file 6-8
- dcmdPeekAtNextChar function 6-12
- dcmdReadRegister function 6-5, 6-14
- dcmdSetPosition function 6-12
- dcmdWriteRegister function 6-5, 6-15
- debugger extensions 6-3 to 6-15
 - building 6-8
 - callback functions 6-14
 - callback routines 6-11 to 6-15
 - dcmdGetNextChar 6-12
 - dcmdGetNextExpression 6-13
 - dcmdGetNextParameter 6-12
 - dcmdGetPosition 6-11
 - dcmdPeekAtNextChar 6-12
 - dcmdSetPosition 6-12
 - summary 6-7
 - defined 6-3
 - displaying help for 6-3
 - executing 6-3, B-38
 - files needed for building 6-8
 - listing 6-3, B-38
 - makefile for building 6-8
 - output from 6-4
 - resource type for 6-4
 - sample code for 6-6
 - skeleton code for 6-5
 - writing 6-4 to 6-7
- debugger host 2-11
- debugger interface
 - changing appearance of 2-12 to 2-14
 - introduced 1-9 to 1-10
- debugger nub 2-11
- Debugger Nub Controls dialog boxes 2-10
- Debugger Nub Controls file 1-8
- Debugger routine
 - disabling calls to 3-15
 - inserting in source code 3-14
- debugger shortcuts C-3 to C-4
- debugger windows, shown on launch 3-3
- DebugStr routine
 - disabling calls to 3-15
 - inserting in source code 3-14
- decimal numbers

- notation used for A-3
- default format, displaying in 3-29, B-29
- Disable Utility command B-33
- disabling calls to debugger 3-15
- displaying breakpoints 3-14
- displaying embedded symbols 2-13, B-7
- displaying memory. *See* memory display

E

- Edit menu B-5 to B-11
 - Clear B-6
 - Copy B-5
 - Cut B-5
 - General Preferences 2-12 to 2-14, B-6
 - Paste B-6
 - Remember Window Position B-11
 - Select All B-6
 - Show Clipboard B-6
 - Symbolic Mapping Preferences 3-33, B-9
 - Undo B-5
- embedded symbols, displaying 2-13, B-7
- emulated code, debugging 1-4
- Enable Utility command B-33
- Enter MacsBug command B-37
- error messages A-9
- Evaluate "" command B-28
- Evaluate menu B-27 to B-29
 - Evaluate "" B-28
 - Evaluate "this" B-29
 - Show Evaluation Window B-28
- Evaluate "this" command B-29
- exception handler, calling from debugger 3-16, B-24
- exceptions 3-16, B-24
- Execute Debugger Extension command B-37
- execution
 - animate mode 3-17, B-26
 - controlling 3-15
 - resuming, of target 3-16, B-24
 - stepping 3-16, B-25
 - stepping into procedure calls 3-17, B-25
 - stepping out of current routine 3-17, B-26
 - stopping, of target B-24
- expression evaluator A-3
- expressions
 - constants in A-3
 - displaying value of 3-28
 - errors returned in evaluating A-9
 - evaluating B-28
 - grammar of A-3, A-5 to A-9
 - precedence of operators A-4
 - use of symbols in A-3
- expression syntax A-5 to A-9

- extended mnemonic set 2-14, B-8
- extensions. *See* debugger extensions
- Extras menu B-36 to B-38
 - Enter MacsBug B-37
 - Execute Debugger Extension B-37
 - Search Memory 3-20, B-36
 - Show PC B-36
 - Snapshot Active Window B-38
 - Stop For DebugStrs B-37

F

- File menu B-3 to B-5
 - Close B-4
 - Map Symbolics to Code B-3
 - Open B-3
 - Open ROM map B-3
 - Page Setup B-4
 - Print Window B-4
 - Quit B-5
 - Save Log As B-4
 - Save Performance B-4
- files needed
 - for 68K Mac Debugger 1-6
 - for low-level debugging 1-8
 - for PowerPC source-level debugging 1-7
- Find Code For "" command B-22
- Find Code For command B-22
- Find Code For dialog box 3-7, B-22
- Fixed-Point Exception Register 5-9
- flat-time measurement 4-4
- floating-point registers 3-26 to 3-27, 5-8, B-16
- Floating-Point Status and Control Register 3-27, 5-9, B-16
- FPSCR. *See* Floating-Point Status and Control Register
- Fragment Info window 3-31
- fragments
 - code section 5-5
 - data section 5-5, 5-7
 - displaying list of 3-31
 - handling 3-30 to 3-33
 - locating in memory 5-5
 - referenced by the system 5-7
 - and symbolic mapping 3-33

G

- Gather Report command B-33
- General Preferences command 2-12 to ??, B-6
- General Preferences options 2-13, B-6
- general-purpose registers 3-25 to 3-26, 5-7, B-14 to B-15

global variables
 displaying value of 3-29, B-13
 special notation used in display of 3-30, B-14
 and unmangle schemes D-3
 glue code 2-13, B-7

H

handling fragments 3-30 to 3-33
 header files for debugger extensions 6-8
 heaps 5-7
 hexadecimal addresses, conversion to 3-19
 hexadecimal numbers, notation used for A-3
 hexadecimal values
 and searching 3-20
 displayed as unsigned long 2-13, B-7
 Hide Control Palette command B-27

I

icons
 breakpoint 3-9
 in control palette 3-16 to 3-17
 for program counter in browser 3-17
 resize 3-7
 informational alerts, displaying 2-14, B-8
 initialization code, debugging B-25
 input routines 6-11
 installation
 of 68K Mac Debugger 2-5
 of Power Mac Debugger, low level 2-8
 of Power Mac Debugger, source level 2-7
 instruction display
 displaying 680x0 instructions 3-22
 displaying PowerPC instructions 3-21
 printing contents of 3-8
 starting address of disassembly 3-22
 instruction window
 selecting mnemonics used 2-14, B-8
 starting address of disassembly B-18
 Integer Exception Register A-4
 interface, debugger. *See* debugger interface

K

Kill "" command B-24

L

Launch command 3-18, B-25
 launching an application 3-18, B-25
 leaf routines 5-11
 library files for debugger extensions 6-8
 Link Register 5-9, A-4
 log window 3-3, 3-8, 6-4, B-4, B-23
 low-level debugging 5-3 to 5-15
 files needed for 1-8
 getting started 2-8 to 2-12
 low-speed serial connection 2-13, B-7

M

MacsBug
 entering from the Enter MacsBug command B-37
 extensions and debugger extensions 6-3, 6-5, 6-7
 and mixed-mode debugging 5-15
 obtaining heap information with 5-7
 optional component of 68K Mac Debugger 1-6
 optional component of Power Mac Debugger 1-7, 1-8
 main process window 2-13, B-7
 manually targeting a running process B-25
 map symbolics to code 3-33, B-3
 Map Symbolics to Code command 3-33, B-3
 memory
 allocation of multibyte values in 5-3
 data alignment in 5-4
 size of 5-3
 memory display 3-18 to 3-27
 beginning address of disassembly 3-19, B-17
 byte grouping, changing 3-20, B-17
 changing values in 3-19, B-17
 interpreting values in 5-3
 of 680x0 instructions 3-22 to 3-23
 of PowerPC instructions 3-21 to 3-22
 printing contents of 3-8
 searching in B-36
 setting default 2-14, B-8
 misaligned data 5-4
 mixed-mode debugging 5-15
 mnemonics
 displaying POWER 2-14, 3-22, B-8
 displaying PowerPC 2-14, 3-22, B-8
 multiple applications, debugging 2-11
 multiple code views 3-6

N

names, displaying value of 3-28

ndcdGlue.o file 6-8
 ndcd.h file 6-8
 'ndcd' resource type 6-4, 6-9
 New Session command B-30
 .NJ files 2-4
 node tables 4-16 to 4-19
 nonvolatile registers 5-7 to 5-9
 NubExt.xcoff file 6-8

O

one-machine setup
 for 68K Mac Debugger 1-6
 for Power Mac Debugger 1-7
 selecting B-8
 one-shot breakpoints 3-11
 Open command B-3
 Open ROM map command B-3
 operators, precedence of A-4
 optimization
 and effect on compiler 2-9
 and source-level debugging 2-4
 output routines 6-13

P

Page Setup command B-4
 Paste command B-6
 PC. *See* program counter
 performance breakpoints 3-13, 4-8
 performance data, evaluating 4-15
 performance document
 creating 4-6, B-30
 printing 4-14
 saving 4-14
 views in 4-6, 4-11 to 4-13, B-32
 performance information
 showing with zero hits 2-13, B-7
 performance measurement. *See* ASP
 Performance menu B-29 to B-36
 Configure Report B-32
 Configure Utility command B-31
 Disable Utility B-33
 Enable Utility B-33
 Gather Report B-33
 New Session B-30
 Show Source B-31
 performance report 4-9 to 4-13
 editing 4-13, B-32
 generating 4-9
 pointer to the Table of Contents A-4

Power Mac Debugger
 low level
 components 1-8
 installing 2-8
 launching 2-10
 source level
 components 1-7
 installing 2-7
 launching 2-8
 Power Mac Debugger Prefs file 1-7, 1-8
 Power Mac DebugServices application 1-7
 POWER mnemonics 2-14, 3-22, B-8
 PowerPC floating-point registers 3-26
 displaying 3-27
 PowerPC general-purpose registers 3-25, 5-7, B-14
 PowerPC mnemonics 2-14, 3-22, B-8
 PowerPC registers
 See also PowerPC floating-point registers; PowerPC
 general-purpose registers
 names of in expressions A-4
 usage of 5-7 to 5-9
 PowerPC stack frame 5-13
 PPC Debugger Nub extension 1-8
 PPCDebuggerNubINIT extension 1-8
 PPCTraceEnabler extension 1-7
 PPCTraceEnabler file 1-8
 Print Window command B-4
 profiling. *See* ASP
 program counter
 displaying 3-17, B-14, B-36
 and expression grammar A-4
 and memory access 5-9
 in subroutine marker 3-17
 program execution. *See* execution
 Propagate Exception command 3-16, B-24
 propagating an exception 3-16, B-24
 Put.c file 6-8
 Put.h file 6-8

Q

Quit command B-5

R

record bit (Rc) 5-8
 recursive routine, displaying local variables in 3-25,
 B-13
 Red Zone 5-11
 register names A-3 to A-4
 registers 3-25 to 3-27

- accessing 5-7
- floating-point 3-26 to 3-27, B-16
- general-purpose 3-25 to 3-26, 5-7, B-15
- names of in expressions A-3
- PowerPC floating-point 3-27
- registers window
 - examples of 3-25 to 3-27, B-14 to B-16
 - introduced 3-3
 - as main process window B-7
- Remember Window Position command B-11
- report configuration dialog box 4-14
- resize icons 3-7
- resonance effect, in profiling 4-15
- resource editors 6-3
- ResourceTracker (68K) E-3
- Resume "" command B-24
- Rez resource compiler 6-3
- RomInfo file 1-7, 1-8
- ROM map files
 - opening B-3
 - selecting 2-14, B-8
- routines, finding source code for 3-6
- ROTC (Table of Contents Register) 5-7, B-15
- Run "" command 3-16, B-24

S

- sampling rate, of ASP 4-3, B-31
- Save Log As command B-4
- Save Performance command B-4
- scratch fields 5-8
- scratch registers 5-8
- Search Memory command 3-20, B-36
- Search Memory window 3-21
- Select All command B-6
- serial connection 2-8, 2-13, B-7
- setting breakpoints from source code 3-14
- shared libraries, measuring performance of 4-4
- shortcuts, for debugger C-3 to C-4
- Show 68K Instructions command B-19
- Show Breakpoints List command B-26
- Show Clipboard command B-6
- Show Control Palette command 3-15, B-27
- Show Evaluation Window command B-28
- Show Exports window 3-32
- Show FPU Registers command B-15
- Show Fragment Info command B-20
- Show Globals command B-13
- Show Instructions command B-18
- Show Log Window command B-22
- Show Memory command B-17
- Show PC command B-36
- Show Registers command B-14

- Show Source command B-31
- Show Stack Crawl command B-11
- Show Tasks command B-19
- simple breakpoints 3-10
- Snapshot Active Window command B-38
- source file, selecting 3-4
- Source File For "" command B-22
- source-level debugging 1-6 to 1-8, 3-3 to 3-33
- special-purpose registers 5-9
- stack
 - 680x0 frame 5-12
 - accessing 5-9 to 5-13
 - direction of growth B-12
 - displaying local variables for a routine B-12
 - interpreting display 5-11
 - pointer B-12
 - PowerPC frame 5-13
 - Red Zone 5-11
 - switch frame 5-11
 - type of frame in B-12
- stack-based calling conventions, Pascal and C 5-12
- stack frame B-12
- stack pointer 5-7, A-4, B-15
- stack pointer, changing 2-14, B-8
- stack window
 - as main process window 2-13, B-7
 - opening B-11
 - resizing panes in 3-23, B-12
 - shortcuts for C-4
 - and Show Stack Crawl command 3-23, B-12
- stand-alone code, measuring performance of 4-4
- statistics view 4-6, 4-11 to 4-13, B-31
- Step command 3-16, B-25
- Step Into command 3-17, B-25
- Step Out command 3-17, B-26
- stepping into procedure calls 3-17, B-25
- stepping out of current routine 3-17, B-26
- stepping through source code 3-16, B-25
- stepping to a branch instruction 3-18
- Step to Branch command 3-18, B-26
- Step to Branch Taken command 3-18, B-26
- Stop "" command 3-16, B-24
- Stop For DebugStrs command B-37
- stopping a targeted process 2-13, B-7
- stub routines 6-7
- summary view 4-6, 4-11, B-31, B-34
- switch frames 5-11
- SYM big option 2-4
- symbolic information files
 - and debugging 1-4
 - opening B-3
 - out of date 2-13, B-7
 - relation to browser window B-4
 - selecting for debugger B-9
 - use of in source-level debugging 2-3 to 2-4

symbolic mapping preferences 3-32
 Symbolic Mapping Preferences command B-9
 symbolics, mapping to code B-3
 symbols
 evaluating B-28
 use of in expressions A-3
 .SYM files 2-4
 system-level debugging 5-14
 System Peripheral 8 Cable, and low-level
 debugging 2-8
 system software, debugging 1-4

T

Table of Contents (TOC) area 5-7, B-15
 Table of Contents Register (RTOC) 5-7
 target application, launching 2-11
 Target "" command 3-18, B-25
 targeted process, stopping 2-13, B-7
 targeting a running process 3-18, B-25
 target machine 2-11
 Target Preferences command B-8
 "this", displaying value of 3-29
 two-machine setup
 for 68K Mac Debugger 1-6
 for Power Mac Debugger 1-7, 1-8
 selecting B-8

U

Undo command B-5
 unmangle schemes, creating D-3 to D-6
 unsigned long values, displayed as hexadecimal 2-13,
 B-7
 Untarget "" command 3-18, B-25
 untargeting a process 3-18, B-25
 user breaks 3-14, 3-15
 utility configuration dialog box 4-7
 utility routines 6-14

V

variables, displaying value of 3-28 to 3-30
 Views menu B-11 to B-23
 Find Code For B-22
 Find Code For "" B-22
 Show 68K Instructions B-19
 Show FPU Registers B-15
 Show Fragment Info 5-6, B-20

 Show Globals B-13
 Show Instructions B-18
 Show Log Window B-22
 Show Memory B-17
 Show Registers B-14
 Show Stack Crawl B-11
 Show Tasks B-19
 Source File For "" B-22
 volatile registers 5-7 to 5-9

W

warnings, source out of date 2-13, B-7
 window position, saving B-11
 windows
 for displaying memory 3-7 to 3-8
 log B-23
 making active B-38
 printing contents of B-4
 saving snapshot of B-38
 selecting contents of B-6
 selecting scrolling action in 2-13, B-7
 Windows menu B-38

X

.xcoff files 2-4
 .xSYM files 2-4

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 Apple Courier.

WRITERS

Joanna Bujes, Jeanne Woodward

EDITOR

Laurel Rezeau

ILLUSTRATOR

Sandee Karr

PRODUCTION EDITOR

Lorraine Findlay

ONLINE PRODUCTION EDITOR

Alexandra Solinski

PROJECT MANAGER

Patricia Eastman

Special thanks to Sandeep Gupta,
Gary Kratzer, Jack Littleton,
Kevin Looney, and Kevin Redden.