

# Xcode Tools Sensei

Your Guide to the Mac OS X Developer Tools



Mark Szymczyk

# Xcode Tools Sensei

Your Guide to the Mac OS X Developer Tools

Mark Szymczyk

Me and Mark Publishing

Published by Me and Mark Publishing  
<http://www.meandmark.com>

Copyright ©2006 by Me and Mark Publishing

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means without written permission from the publisher or author.

ISBN, print ed.	0-9761260-0-1
ISBN, electronic ed.	0-9761260-1-X
Library of Congress Control Number	2004096946

TRADEMARKS — Apple, Mac, Macintosh, Mac OS, Xcode, Carbon, Cocoa, AppleScript, Aqua, eMac, Finder, iBook, iMac, iPod, iTunes, Mac Pro, PowerBook, Power Mac, QuickDraw, QuickTime, Safari, and Velocity Engine are trademarks of Apple Computer, Inc. PowerPC is a trademark of International Business Machines Corporation. OpenGL is a trademark of Silicon Graphics, Inc. Java is a trademark of Sun Microsystems, Inc. Windows is a trademark of Microsoft Corporation. All other products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations.

DISCLAIMER — This book was independently published by Me and Mark Publishing. Apple is not affiliated with this book. Apple does not endorse or support this book.

Printed in the United States of America

Cover designer: *Carlos Camacho*

# Table of Contents

<b>Introduction</b>	<b>22</b>
<b>Chapter 1: Xcode</b>	<b>26</b>
Creating a Project	26
Action Projects	27
Application Projects	27
Audio Units Projects	28
Bundle Projects	28
Command-Line Utility Projects	28
Dynamic Library Projects	29
External Build System Projects	29
Framework Projects	29
J2EE Projects	30
Java Projects	30
Kernel Extension Projects	31
Standard Apple Plug-In Projects	31
Static Library Projects	31
Creating Your Own Project Templates	31
The Project Window	32
Toolbar	33
Groups and Files List	33
Project Name	34
Targets	35
Executables	35
General Panel	35
Arguments Panel	36
Debugging Panel	37
Errors and Warnings	37
Find Results	37
Bookmarks	37
SCM	37
Project Symbols	38
Smart Groups	39
Adding Files and Frameworks to Your Project	40
Adding New Source Code Files to the Project	40
Choosing a File Type	40
Naming the File	40
Creating Your Own File Templates	41
Fixing the Copyright Notice	42
Adding Files You've Already Created	43
Source Trees	44
Adding Frameworks	44

Editing Source Code	44
The Editor Window	44
Toolbar	45
Customizing the Toolbar	45
Status Bar	45
Navigation Bar	46
Editor and Gutter	46
Code Completion	46
Using the Completion List	46
Cycling Through Completion Matches	47
Customizing Code Completion	47
Getting Information About Functions and Variables	48
Customizing Source Code Editing	48
Text Editing Preferences	48
Fonts and Colors Preferences	48
Indentation Preferences	49
Key Bindings Preferences	49
Using Xcode's Class Browser	50
Browsing Classes	50
Customizing What the Class Browser Shows	50
Customizing What Appears in the Class List	51
Customizing What Appears in the Member List	51
Using Xcode's Modeling Tools	51
Modeling Classes	51
Adding a Class Model to Your Project	52
The Class Model Window	52
Class List	53
Member List	53
Selection Area	53
Class Diagram	53
Opening the Information Panel	54
Customizing the Class Diagram	54
Adding Comments	54
Filtering Information from the Diagram	55
Adding and Removing Files to Track	55
Modeling Data	55
Adding a Data Model File to Your Project	56
Data Model Window	56
Entity List	56
Property List	57
Selection Area	57
Diagram	57
Adding Entities	58
Adding Attributes	58
Adding Relationships	58

## 6 Table of Contents

Adding Fetched Properties and Fetch Requests	59
Editing Predicates	59
Setting Information Dictionary Entries	60
Adding Configurations	60
Creating Source Code	60
Reading Developer Documentation	60
Searching the Developer Documentation	61
Search Groups List	61
Bookmarks List	61
Detail View	61
Editor View	62
Working with Targets	62
Adding Targets	62
Special Targets	63
BSD Targets	63
Carbon Targets	63
Cocoa Targets	63
Java Targets	63
Kernel Extension Targets	63
Legacy Targets	64
Unit Testing Targets	64
Duplicating Targets	64
Target Build Phases	65
Adding Build Phases to a Target	65
Reordering Build Phases	66
Moving a File to a Different Build Phase	66
Inspecting Target Settings	66
General Panel	66
Build Panel	67
Rules Panel	68
Properties Panel	68
General Property Settings	69
Cocoa-Specific Settings	69
Document Types	69
Configuring the Compiler	70
Build Configurations	70
Build Styles	72
Build Settings	72
General Settings Collection	73
Architectures	73
Build Locations	73
Search Paths	74
Versioning	74
Build Options	74
Linking	74

Packaging	75
Deployment	75
Unit Testing	75
GNU C/C++ Compiler Settings Collection	75
Language	76
Code Generation	77
Warnings	77
Preprocessing	78
Rez Settings Collection	78
Lex Scanner Generator Settings Collection	78
Yacc Parser Generator Settings Collection	78
Adding Your Own Build Settings	79
Setting Compiler Flags for One File	79
Java Compiler Settings	80
Configuration Files	81
Creating a Configuration File	81
What Goes in a Configuration File?	81
Telling Your Project to Use a Configuration File	82
Overriding the Configuration File	82
Compiling Your Program	83
Precompiled Headers	83
ZeroLink	83
Distributed Builds	84
Cleaning Targets	85
Building Your Project	85
Seeing More Build Details	85
Building for Unsupported Languages	86
Tips for Correcting Build Errors	87
Add All Necessary Frameworks	87
Include Necessary Header Files	87
The Error May Not Be Where Xcode Says It Is	87
One Error Can Cause Multiple Syntax Errors	87
Look for Typographical Errors	88
Check Function Arguments	88
Running your Program	88
Developing for Different Versions of Mac OS X	89
Choosing a SDK	89
Choosing a Deployment Target	89
Supplying a Prefix File	90

## 8 Table of Contents

Creating Universal Binaries for PowerPC and Intel Hardware	90
Installing the Mac OS X 10.4 Universal SDK	90
Selecting gcc 4 as the Compiler	91
Using the Mac OS X 10.4 Universal SDK in Your Project	91
Setting the Deployment Target to Mac OS X 10.4	91
Building for PowerPC and Intel Architectures	91
Building Universal Binaries for Older Versions of Mac OS X	92

## **Chapter 2: Debugging with Xcode** **93**

Before You Debug	93
Configuring Xcode for Debugging	93
Choosing a Debugging Format	94
Setting Debugging Options for Your Program	94
Standard Input/Output	95
Remote Debugging	95
Start Executable After Starting Debugger	95
Break on Debugger() and DebugStr() Calls	95
Adding Places to Look for Files	95
Setting Environment Variables for Debugging	96
Using the Debug Version of Frameworks	96
Guard Malloc	96
Enabling Remote Debugging	98
Allowing Remote Login	98
Generating ssh Keys	98
Creating a Shared Folder for the Project's Build Products	99
Turning on Remote Debugging	99
Launching the Debugger	100
Call Stack Viewer	100
Variable Viewer	100
Custom Data Formatters	101
Viewing Shared Libraries	102
Viewing Global Variables	103
Viewing Registers	103
Breakpoints	103
The Breakpoints Window Before Xcode 2.1	105
The Breakpoints Window After Xcode 2.1	106
Setting Watchpoints in Xcode	106
Stepping Through Your Code	107
Viewing Memory	107
Looking at Disassembled Code	108
Fixing Your Code While Debugging	108
Debugging Command-Line Programs	109



Using the gdb Console	109
Stopping Program Execution	110
Setting Breakpoints	110
Setting Watchpoints	111
Setting Catchpoints	111
Examining Your Breakpoints	111
Setting Conditional Breakpoints	112
Disabling and Deleting Breakpoints	113
Command Lists	114
Examining Data	115
Examining Dynamic Arrays	115
Displaying Data Automatically	115
Executing Shell Commands	117
Defining Your Own Commands	117
Conditional Commands	118
Documenting Your Commands	119
Reading Commands from a File	120
Command Hooks	121
 <b>Chapter 3: Interface Builder</b>	 <b>122</b>
Starting with Interface Builder	122
Creating User Interfaces for Cocoa Programs	123
Laying out the Interface	123
Adding Palettes to the Palettes Window	124
Address Book Palette	124
AppleScript Palette	124
Cocoa Menus Palette	124
Cocoa Controls and Indicators Palette	125
Cocoa Text Controls Palette	125
Cocoa Windows Palette	126
Cocoa Data Views Palette	127
Cocoa Container Views Palette	127
Cocoa Graphics Views Palette	128
Controllers Palette	128
Automator Palette	129
Disc Recording Palette	129
OSA Palette	129
PDF Kit Palette	129
UIKit Palette	129
Quartz Composer Palette	130
Customizing the Palette Window Toolbar	130

## 10 Table of Contents

Modifying the Interface	130
Attributes Panel	131
Connections Panel	131
Size Panel	132
Autosizing	132
Size Panel for Windows	133
Auto Positioning Windows	133
Bindings Panel	134
Custom Class Panel	134
Accessibility Panel	135
Help Panel	135
AppleScript Panel	135
Sherlock Panel	136
Creating Source Code in Interface Builder	136
Creating Subclasses	137
Adding Outlets and Actions	137
Creating Instances in the Nib File	138
Creating Source Code Files	138
Parsing Xcode Header Files into Interface Builder	139
Making Connections	139
Creating Bindings	139
Creating the Model Class	140
Creating the Controller	140
Binding the Model to the Controller	140
Binding the View to the Controller	141
Value Transformers	141
Placeholders	142
Working with Menus	142
Adding Menus to the Menu Bar	142
Adding Menu Items	142
Creating a Dock Menu	143
Creating a Contextual Menu	143
Testing Your Interface	144
The Nib File Window for Cocoa Applications	144
Instances Tab	144
File's Owner	145
First Responder	145
Outline View	145
Classes Tab	146
Images Tab	147
Sounds Tab	147
Nib Tab	147
Creating Cocoa Nib Files	148

Creating User Interfaces for Carbon Programs	149
Laying out the Interface	149
Menus Palette	149
Controls Palette	150
Enhanced Controls Palette	150
Browsers and Tab Palette	151
Windows Palette	152
Text Based Controls Palette	152
Modifying the Interface	152
Attributes Panel	153
Control Panel	153
Size Panel	154
Layout Panel	154
Help Panel	155
Working with Menus	155
Adding Menus to the Menu Bar	155
Adding Menu Items	155
Setting a Menu Item's Command	156
Creating a Dock Menu	156
Creating a Contextual Menu	157
The Nib File Window for Carbon Applications	157
Instances Tab	157
Images Tab	158
Nib Tab	158
Importing Resource Files	158
Creating Carbon Nib Files	159
 <b>Chapter 4: Sampler</b>	 <b>160</b>
Call Stacks	160
Running Sampler	161
Profiling Your Program	161
Examining the Results	162
Outline View	164
Trace View	165
Examining Memory Allocations	166
Examining Specific Functions	167
Options to Control Sampler's Output	168
Generating Reports	169
Call Graph Report	170
Function Cross Reference	170
Library Cross Reference	170

<b>Chapter 5: gprof</b>	<b>171</b>
Generating Profiling Code In Xcode	171
Running gprof	172
Interpreting gprof's Results	174
Flat Profile	174
Call Graph Profile	175
Cycles of Recursion	177
gprof Options	177
-a Option	177
-b Option	178
-e Option	178
-E Option	178
-f Option	178
-F Option	179
-s Option	179
-S Option	179
-z Option	180
 <b>Chapter 6: CHUD Tools</b>	 <b>181</b>
Shark	181
Configuring Shark	181
Time Profiles	182
System Trace	183
Function Trace	183
Java Traces	183
Data Cache Miss Profiles	183
Malloc Trace Profile	184
Memory Bus Bandwidth Profiles	184
Static Analysis Profile	184
VM Faults Profile	184
Windowed Time Facility	185
Setting the Sampling Rate	185
Recording Performance Events	185
Advanced Configuration Options	186
Choosing a Trigger	186
Choosing the CPU and Memory Controller	187
Filtering Programs to Profile	187
Setting Events to Record	188
G5 Specific Events	188
Event Multiplexing	188
Setting Intel Performance Events	191
Creating Profiling Equations	192
Configuring the Profiling Session	192
Profiling Your Program	193

Viewing Shark's Results	193
Heavy and Tree Views	194
Showing the Call Stack Table	194
Viewing Source Code	195
Customizing the Code Browser	195
Viewing Assembly Language Code	195
Viewing High-Level Language Code	197
Customizing What the Results Window Displays	198
Viewing Charts	198
Call Stack Chart	199
Performance Event Charts	199
Viewing Individual Samples	199
Data Mining	200
Excluding Portions of Code	200
Excluding Areas of Code	200
Excluding Individual Functions and Libraries	201
Flattening Libraries	201
Focusing on Functions	201
Performance Event Data Mining	202
System Trace Results	202
Summary	202
Scheduler Summary	203
System Calls and VM Faults Summaries	203
Trace	203
Scheduler Trace	204
System Calls Trace	204
VM Faults Trace	204
Timeline	205
MONster Profile	205
Saturn	206
Before You Profile	206
Profiling Your Program	206
Viewing Saturn's Results	207
CHUD Command-Line Tools	207
amber	208
simg4	209
Output	209
Instruction Flow Statistics	209
Dispatch Stalls	210
Execution Unit Statistics	210
Retirement Stalls	211
Branch Statistics	211
L1 Instruction Cache Statistics	213
ITLB Statistics	213
L1 Data Cache Statistics	214

## 14 Table of Contents

DTLB Statistics	214
Software Prefetching Statistics	215
L2 Direct Mapped SRAM Statistics	215
L2 Cache Statistics	216
Processor Bus Statistics	217
Pipe Output	219
Horizontal Scroll Pipe	219
Vertical Scroll Pipe	219
simg5	220
Output	220
CPI	221
Branch Prediction Statistics	222
Instruction Fetch and Translation Statistics	222
Instruction Cache Statistics	223
Data Side Translation Statistics	223
Data Prefetch Statistics	223
Data Cache Statistics	224
Execution Unit Statistics	224
Queue Resource Usage Statistics	224
Rename Resource Usage Statistics	225
Instruction Frequency	225
CPI Stack	226
Pipe Output	226
acid	227
Summary Information	228
Instruction Statistics	228
Use Distance Statistics	228
Branches	229
Data Access Statistics	229
Stall Cycles	230
Execution Serializing Instructions	230
Misaligned Accesses	230
Most Used Register List	231
Detailed Statistics	231
Instruction Mix Statistics	231
Executed Instruction List	231
Stall Cycles	232
Target-Use Distance Counts	232
Load-Use Distance Counts	232
Basic Block Length Counts	233
Register Use Statistics	233
Data Address Statistics	233
Instruction Address Statistics	233

Gathering Data on Each Instruction	234
-a Option	234
-A Option	234
<b>Chapter 7: MallocDebug</b>	<b>235</b>
Running MallocDebug	235
Examining Your Program's Memory Usage	236
Flat View	237
Inverted View	237
Examining Individual Memory Allocations	237
Finding Memory Leaks	238
Finding Memory Overruns and Underruns	238
Inspecting Memory Zones	239
Narrowing Your View	239
Viewing Recent Memory Usage	239
Pruning Functions from the Call Tree	239
Organizing Memory Allocations with Mappings	240
<b>Chapter 8: ObjectAlloc</b>	<b>241</b>
Running ObjectAlloc	241
Global Allocations View	242
Instance Browser View	244
Call Stacks View	245
Stepping Through Allocations	246
Finding a Specific Memory Event	246
Showing Fresh Memory Allocations	247
<b>Chapter 9: Command-Line Debugging Tools</b>	<b>248</b>
A Command Line Primer	248
Executing Commands as root	248
Navigating Directories	248
Getting Help	249
Finding Your Application's Process ID	250
fs_usage	250
Reporting File Manager Routines	250
Running fs_usage	251
What fs_usage Tells You	251
fs_usage Options	253
-e Option	253
-f Option	253
-w Option	253

## 16 Table of Contents

sc_usage	254
What sc_usage Tells You	254
Program Summary Information	255
System Call List	255
Blocked System Call List	256
sc_usage Options	256
-c Option	256
-e Option	257
-E Option	257
-l Option	257
-s Option	257
vmmap	258
What vmmap Tells You	258
Non-Writable Memory Regions	258
Region Purpose	259
Permissions	259
Sharing Modes	260
Writable Memory Regions	260
Summary Report	261
vmmap Options	262
-d Option	262
-w Option	262
-resident Option	262
-pages Option	263
-interleavedOption	263
-submap Option	263
-allSplitLibs Option	263
heap	264
leaks	265
Running leaks	265
What leaks Tells You	265
leaks Options	266
-nocontext Option	266
-nostacks Option	266
-exclude Option	266
malloc_history	267
Running malloc_history	267
Running malloc_history on a Specific Memory Area	267



<b>Chapter 10: gcov</b>	<b>268</b>
Generating Code Coverage Data in Xcode	268
Running gcov	269
Interpreting gcov's Results	270
Running Multiple Tests	270
gcov Options	270
-a Option	270
-b Option	271
-c Option	272
-f Option	272
-l Option	272
-n Option	272
-o Option	272
-p Option	273
-u Option	273
 <b>Chapter 11: Version Control with cvs</b>	 <b>274</b>
What You Must Do From the Command Line	274
Setting the \$CVSROOT Environment Variable	274
Creating a Repository	275
Letting Other People Access the Repository	275
Dealing with Binary and Bundled Files	276
Accessing a Remote Repository Using ssh	277
Telling cvs to Use ssh for Remote Access	277
Setting \$CVSROOT to a Remote Repository	278
Generating Keys	278
Moving Your Key to the Repository	279
Adding a Project to the Repository	280
Checking Files Out of the Repository	280
Using cvs in Xcode	281
Turning on Version Control	281
Did the Command-Line Checkout Succeed?	281
Seeing Which Files Aren't Up To Date	282
Committing Changes You Made	283
Discarding Changes You Made to a File	283
Adding Files to the Repository	283
Removing Files from the Repository	283
Seeing a File's SCM Information	284
Comparing Two Versions of a File	284
Reverting to an Old Version of a File	284
Viewing cvs Annotations	284

<b>Chapter 12: Java Tools</b>	<b>285</b>
Jar Bundler	285
Setting Build Information	285
Options for Main	286
Custom Icon	286
JVM Version	286
Use Macintosh Menu Bar	287
Anti-alias Text and Graphics	287
Growbox Intrudes	287
Disable .app Package Navigation	287
Live Resizing	287
Enable Hardware Acceleration	287
Smaller Tab Sizes	288
Adding Files to the Bundle	288
Setting Properties	288
Type	288
Signature	289
Heap Size	289
Version	289
Identifier	289
Get-Info String	289
Short Version	289
VM Options	289
Allow Mixed Localizations	290
Development Region	290
Bundle Name	290
Info Dictionary Version	290
Set Working Directory Inside Application Package	290
Additional Properties	290
JavaBrowser	291
The Browser Window	291
Viewing a Class's Description	292
Viewing a Class's Source Code File	292
Viewing a Class's Documentation	292
Searching for Information	292
Filtering Class Information	293
Adding Your Own Classes to the Browser	294
Applet Launcher	294
Launching an Applet	294
Getting Applet Information	295
Serializing Applets	295

<b>Chapter 13: OpenGL Tools</b>	<b>296</b>
OpenGL Profiler	296
Choosing a Program to Profile	296
Custom Pixel Formats	297
Choosing a Graphics Card Driver	297
Setting Environment Variables	297
Remote Profiling	298
Setting Breakpoints	298
Profiling Your Program	299
Viewing the Profiling Data	299
Trace Window	299
Buffers Window	299
Resources Window	300
Scripts Window	301
Breakpoints Window	301
Statistics Window	302
Pixel Format Window	302
Messages Window	302
OpenGL Driver Monitor	303
Getting Started	303
Customizing the Graph	303
Table View	304
Renderer Info	304
OpenGL Shader Builder	305
Writing a Shader	305
Starting and Ending a Shader	306
Placing Comments	306
Declaring Variables	306
Attribute Variables	307
Program Parameter Variables	308
Temporary Variables	314
Result Variables	314
Using OpenGL Shader Builder	315
Giving Your Variables Initial Values	316
Applying a Texture Map	316
GLSL Log Window	316
Debugging Shaders	316
Moving Your Shader to Your OpenGL Program	317

# Acknowledgements

This is the part of the book that is more interesting to people who know me than for people purchasing this book. I get to show people I'm not a self-centered, egotistical jerk by expressing gratitude to the people who helped me with this book.

Even though Apple is not affiliated with this book, they did help me. When my developer mailing with Mac OS X 10.4 did not arrive, the Apple Developer Connection let me download it, which saved me a significant amount of time. David Gleason helped get me answers to technical questions about the Xcode Tools. Sue Carroll helped me get permissions to use screenshots of the Xcode Tools in my book. Thank you to everyone at Apple who helped.

I'd like to thank John Griswell from IBM for answering some questions I had about the `simg5` tool. I'd also like to thank everyone who answered questions on numerous mailing lists and message boards, both the questions I directly asked, and the questions they answered for other people so I didn't have to ask them.

To those of you who read the review copies of my chapters and offered feedback, I thank you. There are too many people to mention individually, but you helped make this a better book.

Thanks to Carlos Camacho for designing the cover of this book. If any of you are thinking about writing and publishing a book, I would recommend getting Carlos to design the cover. He also runs the Mac development sites iDevGames and iDevApps. If you want to learn more about writing Mac software, you should visit his sites.

Finally I'd like to thank my family for their support.

To my brother Steve, thanks for the Cavaliers tickets. If any of you reading this ever get the chance to see LeBron James play live, don't pass it up. If this book sells enough copies, I can make it up to you.

To my brother Dave and sister-in-law Rheia, thank you for your moral support from Austin, Texas.

To my sister Kathy and brother-in-law John, thanks for letting me come over and sit in your air-conditioned house when the weather got too hot.

To my nephews Zachary and Christian, thanks for taking my mind off the book. Playing games like wiffleball, soccer, tag, hide and go seek, animal forget, onion on the loose, and seamonster put an end to several cases of writer's block.

To my niece Alyssa, you weren't born until I had written most of the book, but holding you helped me relax during the stressful time of finishing the book.

And to my parents Stan and Mary, thank you for everything else. This book would not have been possible without your support.



# Introduction

Mac OS X is a great operating system for software developers. Every copy of Mac OS X comes with a set of developer tools, the Xcode Tools. The Xcode Tools contain everything you need to create Mac OS X applications. As powerful as the Xcode Tools are, they can be difficult to learn. Most Mac OS X development books focus on Cocoa programming. They teach enough Xcode and Interface Builder for you to create the projects in the book. But there's more to writing real Mac OS X programs. Real Mac OS X programs must be tested and debugged to make sure they run properly and must be profiled to make sure they run fast enough. A Cocoa programming book isn't going to show you how to profile your program or find memory leaks in it.

*Xcode Tools Sensei* picks up where other books leave off. It teaches the Xcode Tools, not a particular language or programming framework. By reading this book you can spend more time writing, debugging, and profiling your programs and less time searching and reading documentation.

## Contents

26 developer tools. 13 chapters. There's something for every Mac OS X developer in *Xcode Tools Sensei*.

## Xcode

The first two chapters of this book cover Xcode, the centerpiece of the Xcode Tools. Xcode is an integrated development environment (IDE) for writing, compiling, and debugging Mac OS X programs. Xcode 2.0 added visual modeling tools for classes and data structures. In Chapter 1 you'll learn the following tasks:

- Creating an Xcode project for your program.
- Adding files to the project.
- Creating project and file templates.
- Editing source code files.
- Using Xcode 2's visual modeling tools.
- Searching developer documentation.
- Configuring compiler settings.
- Compiling programs.
- Creating universal binaries that run on both PowerPC and Intel processors.

Chapter 2 covers Xcode's debugging capabilities. In this chapter you'll learn how to pause your program, execute one line of code at a time, and view your program's variables. Use Xcode's debugger to make sure your program runs the way you expect it to run and to find mistakes in your code.

## Interface Builder

Chapter 3 covers Interface Builder, which is the tool to build user interfaces in Mac OS X. In this chapter you'll learn how to create user interfaces for Cocoa and Carbon applications. You'll also learn to perform the following tasks for Cocoa applications:

- Connect items so they can communicate with each other in your program.
- Create Cocoa subclasses.
- Add functions to Cocoa classes.
- Use bindings to manage relationships between user interface elements and data.

## Performance Tools

Mac OS X programs must run fast enough to be responsive to the user. Unresponsive applications are at best frustrating and at worst unusable. If your program runs too slow, you must find the slow areas of the program so you can make the program run faster. The next three chapters cover tools that can find the slow spots in your code.

Chapter 4 covers Sampler, which is a tool that examines your program's call stack periodically. By viewing the call stack statistics, you can discover the functions in your program where the program spends the most time.

Chapter 5 covers `gprof`, which is a command-line profiler. You'll learn how to tell Xcode to generate the profiling information `gprof` requires, how to run `gprof`, and how to view `gprof`'s results.

Chapter 6 covers the CHUD Tools, which are Mac OS X's most powerful profiling tools. The chapter covers the graphical programs Shark and Saturn as well as the command-line tools `amber`, `simg4`, `simg5`, and `acid`.

Profiling your program with Mac OS X's performance tools is only half the battle. Interpreting the data the tools generate is the other half of the battle. Read Chapters 4–6 to win the whole battle.

## Memory Tools

The next two chapters cover tools that help you find memory-related problems in your programs. Chapter 7 covers MallocDebug, which lets you see how much memory you're allocating in each function. It also detects memory leaks and memory overruns, where you write past a block of memory. Chapter 8 covers ObjectAlloc, which is a tool that lets you examine every memory allocation your program makes.

## Command-Line Debugging Tools

Chapter 9 covers the Unix command-line debugging tools `fs_usage`, `sc_usage`, `vmmap`, `heap`, `leaks`, and `malloc_history`. If you've never heard of these tools before, don't worry. After reading Chapter 9 you'll become intimately familiar with them.

### **gcov**

Chapter 10 covers `gcov`, which measures how many times your program executes each line of code in your program. By using `gcov` you can ensure that your program executes every line of code in your program.

### **cvs**

Chapter 11 shows you how to work with version control using `cvs`. Version control lets you see the changes you make to source code files and lets you go back to previous versions of files. After reading Chapter 11 you'll know how to perform the following tasks:

- Create a repository to store your program's files under version control.
- Setup a repository so other developers can access it.
- Import Xcode projects into the repository.
- Add files to the repository from Xcode.
- See the changes you've made to files.

### **Java Tools**

Chapter 12 covers the Java tools. The chapter begins with Jar Bundler, a tool that creates a Mac OS X application bundle out of a Java application. Use Jar Bundler to create an application the user can launch from the Finder. The chapter then covers JavaBrowser, a tool to view Java documentation, and ends with Applet Launcher, a tool to test Java applets.

### **OpenGL Tools**

I finish the book by covering the OpenGL developer tools. There are three OpenGL developer tools.

- OpenGL Profiler, which profiles and debugs OpenGL applications.
- OpenGL Driver Monitor, which displays realtime statistics about your graphics card.
- OpenGL Shader Builder, which you use to create OpenGL shaders to give your OpenGL applications more control over drawing a scene.

## **What About AppleScript Studio?**

If you've visited Apple's developer site, you may have read about AppleScript Studio. The site describes AppleScript Studio as a tool to quickly create Mac OS X applications in AppleScript using the Cocoa framework. But AppleScript Studio does not appear in the table of contents. Why didn't I write about such an important tool?

AppleScript Studio is not an application so I can't include it as a developer tool. AppleScript Studio consists of Xcode, Interface Builder, the Cocoa framework, and AppleScript. Read the Xcode and Interface Builder chapters if you want to learn AppleScript Studio.



## What the Reader Needs to Know

I assume the reader has some experience writing Mac OS X programs. Xcode has built-in support for five programming languages: AppleScript, C, C++, Java, and Objective C. Apple has the Cocoa and Carbon frameworks for developing Mac OS X applications and supports the Swing framework for developing cross-platform GUI applications in Java. I could not cover all these topics adequately and explain the Xcode Tools in one book.

If you're new to programming, I recommend Dave Mark's *Learn C on the Macintosh*. This electronic book complements the book you're reading right now. *Learn C on the Macintosh* teaches the C programming language using Xcode. After learning C, you'll be ready to tackle a Cocoa programming book and start writing Mac OS X applications.

## Some Things to Keep in Mind as You Read This Book

Apple is constantly changing the Xcode Tools. The constant change means Apple is making lots of improvements to the Xcode Tools, which is good. The bad side of constant change is that the way to perform a task can change. I wrote this book for Xcode Tools 2.4. If you have a different version of the Xcode Tools, the screenshots may look different and the way you perform tasks may differ than what appears in the book. Xcode and Shark are the two tools that change the most. You shouldn't have much of a problem with the other tools.

The Xcode Tools have multiple methods to perform many tasks. Rather than detail each possible way to accomplish a task, I usually mention only one of the methods in the chapter text. Just because I mention one way to do something doesn't mean the other methods are worse. You may find it easier to complete a task differently than the approach I mention.

For the latest updates go to this book's official website.

<http://www.meandmark.com/xcodebook.html>

If you have any questions or comments about the book, feel free to email me at the address below. I'll do my best to answer any questions.

[xcodebook@meandmark.com](mailto:xcodebook@meandmark.com)

# Chapter 1

## Xcode

Many computer books use Chapter 1 to cover introductory material. *Xcode Tools Sensei* is not one of those books. I want you to start learning immediately. After reading this chapter you'll know how to create a project, add files to your project, edit source code, model data, read developer documentation, configure the compiler, and build your project into a working Mac OS X program.

### Creating a Project

Every program you write with Xcode requires a project, no matter how small the program is. An Xcode project contains your program's source code files and other files Xcode needs to build a working program, such as Interface Builder nib files. To create an Xcode project, choose File > New Project, and the Project Assistant window opens. Xcode has the following project categories:

- Action
- Application
- Audio Units
- Bundle
- Command-Line Utility
- Dynamic Library
- External Build System
- Framework
- J2EE
- Java
- Kernel Extension
- Standard Apple Plug-Ins
- Static Library

I will go into greater detail on the types of projects shortly, but most of you will be making application projects. After choosing the type of project you want to make, click the Next button. Tell Xcode the name of your project and where you want to store it, then click the Finish button. Congratulations! You've created an Xcode project.

What Xcode includes in a project depends on the type of project you create. Xcode includes the following files for a Cocoa application:

- An Objective C source code file, `main.m`.
- The Cocoa, AppKit, and Foundation frameworks. The AppKit framework contains the user interface portions of Cocoa, and the Foundation framework contains the base classes from which the AppKit classes inherit.
- A nib file, `main.nib`, containing the user interface. Use Interface Builder to modify the interface.
- Two property list files, `Info.plist` and `InfoPlist.strings`. Property list files are XML files that consist of key/value pairs. The key stores the property name, and the value stores the property's value. The `Info.plist` file contains configuration information for the application. The `InfoPlist.strings` file contains localized configuration information. There will be one `InfoPlist.strings` file for each spoken language your application supports.
- A prefix header, `ProjectName.pch`, where *ProjectName* is the name of your project. Xcode precompiles the header files in the prefix file, which makes your project compile faster.

Each project type includes its own set of files. A Carbon application is going to have the Carbon and Application Services frameworks instead of the Cocoa, AppKit, and Foundation frameworks. Java applications (the ones in the Java section of the Project Assistant Window) won't have any nib files and will have a Java source code file. Who would have thought that Xcode would do something as crazy as adding a Java source file to a Java project? What's important to know is each of Xcode's project types provides a starting point for writing your own programs. You can focus on writing code rather than worrying about forgetting a framework.

Assuming you chose an application project, you can run it without writing any additional code. If you're getting antsy to do something on the computer, you can test my previous statement by choosing Build > Build and Run. Xcode will compile the source code and run the application, which displays a window on the screen.

## Action Projects

Action projects create Automator actions. An *action* is a loadable bundle that performs one task. Create workflows by linking actions together. Use workflows to automate tasks in your application and other applications. You could use Automator with iTunes to transfer songs from a CD to an iPod. Apple introduced Automator in Mac OS X 10.4; older versions of Mac OS X cannot use Automator.

You can write Automator actions using AppleScript, Objective C, or shell scripting. Shell script action projects were introduced in Xcode 2.2. Xcode 2.2 also added the definition bundle project. Definition bundle projects allow you to define new data types for Automator actions.

## Application Projects

The most common projects you will be creating will be application projects. Application projects create an executable file you can launch from the Finder. The type of application project you choose depends on the language you want to use. You can create Carbon applications in C or C++, or you can create Cocoa applications using AppleScript, Java, or Objective C. Apple stopped adding features to the Java version of Cocoa in Mac OS X 10.4 so I would advise against writing new Cocoa applications in Java. Improvements Apple makes to Cocoa will not make their way into the Java version.

For Cocoa applications you have the option of writing an application or a document-based application. The difference between the two application types is document-based applications can have multiple documents open at one time. A word processing program is an example of a document-based application; you can have five documents open at once. iTunes is an example of an application; you can't open five windows with each one playing a different song. If your application creates a new window when the user chooses File > New, you should create a document-based application.

Core Data application projects are Cocoa applications that use the Core Data framework. The Core Data framework makes creating data structures for your Cocoa applications easier. Core Data lets you use Xcode's modeling tools to define your program's data structures instead of writing code. Apple introduced Core Data in Mac OS X 10.4; older versions of Mac OS X can't use Core Data.

## Audio Units Projects

Xcode 2.2 added audio units projects. The projects let you create Core Audio audio units. *Audio units* are software components that work with audio data. Audio units let you create plug-ins for audio applications and create your own sound effects.

Xcode provides an audio unit effect component project that creates an audio unit. There are two additional projects that include a view with the audio unit, one using Cocoa and the other using Carbon.

## Bundle Projects

Bundles are directories of files that appear to the user as one file. They can contain executable files, images, sounds, strings, resource files, nib files, libraries, and frameworks. The beauty of bundles is that you don't have to worry about the user deleting or renaming files your program needs to run.

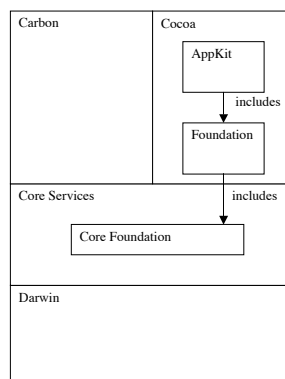
If you look at the Mac OS X documentation included with the Xcode Tools, you'll discover that applications and frameworks are also bundles. This information can cause confusion. When would you use a bundle project instead of an application or framework project? You use bundle projects most often to create plug-ins, programs other applications use to add features to the application. Lots of applications support plug-ins, including Web browsers (Safari), graphics programs (Photoshop and Maya), and development environments (REALbasic).

You can create bundles using the Carbon, Cocoa, and Core Foundation frameworks. Some versions of Xcode split the bundle projects into Bundle and Loadable Bundle sections.

## Command-Line Utility Projects

Command-line utility projects create programs that run from the command line instead of the Finder. If you're learning C or C++, command-line utility projects are perfect. They let you use the standard C and C++ functions to print output on the screen. Those functions don't work with graphical user interfaces like Aqua. By using a command-line utility project, you can learn the language without having to deal with the complexity of writing Mac programs. The Standard Tool project creates a C language tool while the C++ Tool project type creates a C++ tool.

You can also create command-line utility projects that use the Core Services, Core Foundation, and Foundation frameworks. The lowest level of Mac OS X is Darwin, which is a flavor of Unix. The Core Services framework sits on top of Darwin, and higher-level frameworks like Carbon and Cocoa sit on top of Core Services. Figure 1.1 shows the relationship between the frameworks. Core Services contains the operating system services that have nothing to do with a program's user interface, such as networking, file management, memory management, and multiprocessing.



**Figure 1.1**

The relationship between the basic Mac OS X frameworks. The AppKit framework's header file includes the Foundation framework's header file. The Foundation framework's header file includes the Core Foundation framework's header file.

One part of Core Services is the Core Foundation framework, which defines basic data types, such as numbers, strings, arrays, and dates. Core Foundation provides support for plug-ins, bundles, user preferences, property lists, and XML. Because Core Services includes Core Foundation, you can use a Core Services tool project to write Core Foundation code. Only use a Core Foundation tool project if you're sure you don't need to use other parts of the Core Services framework.

Foundation tool projects use the Objective C Foundation framework that is part of the Cocoa framework. Use a Foundation tool project to write an Objective C program without a graphical user interface.

## Dynamic Library Projects

Dynamic libraries are pieces of code an application loads when it's running. You can create dynamic libraries for Mac OS X using the Carbon and Cocoa frameworks. The BSD dynamic library project lets you write a dynamic library using the BSD Unix APIs. If you're writing a dynamic library you want to use on Linux or Unix operating systems, create a BSD dynamic library project.

## External Build System Projects

External build system projects (Older versions of Xcode call them GNU make projects) use a program other than Xcode to build the projects. Use an external build system project when writing a program in a language other than the ones Xcode natively supports: AppleScript, C, C++, Java, and Objective C. An external build system project lets you use Xcode to write code in any programming language, as long as you install that language's compiler on your Mac.

Suppose you're writing a Python program. You can write all your Python code in Xcode and build the program from Xcode by creating an external build system project. All you have to do is tell Xcode to build your program with `pythonw`, the Python interpreter. When you build your project in Xcode, it uses `pythonw` to do the building, letting you avoid the command line.

## Framework Projects

Frameworks are a special kind of bundle. They include header files, resource files, dynamic shared libraries, and reference documentation. Unlike other types of bundles, frameworks can include multiple versions in a single bundle. By storing multiple versions in a bundle, a developer can make improvements to a framework without breaking applications that use an older version of the framework.

Use framework projects if you want other programmers to use your code when creating their applications. Suppose you wrote a class library to handle networking between different operating systems. By writing your class library as a framework project, other programmers could use your code to handle networking in their programs. All they would have to do is add your framework to their projects and include the framework's header files in their source code files.

## J2EE Projects

J2EE projects use the Java 2 Platform, Enterprise Edition (J2EE). Use J2EE to develop component-based large-scale enterprise applications in Java, including applications that run on the Internet. There are three types of J2EE projects.

- Enterprise Java Beans (EJB) module projects create J2EE business components.
- Enterprise application projects create J2EE business applications, which can consist of multiple EJB and Web modules.
- Web module projects create Web applications.

All three types of J2EE projects use Ant to build the project and use the XDoclet code generation engine.

## Java Projects

The main advantage of using Java to write programs is the same code runs on multiple operating systems. Xcode lets you write Java programs that will run on Mac OS X as well as other operating systems like Windows and Linux. Xcode provides many options for writing Java programs.

- Ant projects use Ant to build the program instead of the Java compiler. Ant is a cross-platform build tool that lets programmers build Java programs on any operating system and IDE that supports Ant, such as Xcode. Ant makes open source projects easier to build because it works on multiple operating systems and development environments.
- Abstract Window Toolkit (AWT) projects use the AWT framework. AWT is a class library for writing Java programs with graphical user interfaces.
- Swing projects use the Swing framework to write Java programs with a graphical user interface. Swing inherits from the AWT library, which means the Swing library includes the AWT library as well.
- Java Native Interface (JNI) projects let you access code written in other programming languages, such as C, C++, or Objective C. If you wrote some non-Java code you wanted to use in a Java program, use a JNI project.
- Java tool projects produce programs without a graphical user interface. A Java tool project is the right choice if you're learning Java.

If you use AWT or Swing, you can create applets or applications. Applets are programs that will run in another application. Many websites use Java applets that run in your web browser. Java applications are no different than other Mac OS X applications. You can place them in the Dock and launch them from the Finder.

Assuming you want to write a program with a graphical user interface that runs on multiple operating systems, you must decide whether to create a Swing project or an AWT project. In most cases you'll want to create a Swing project. Because Swing is built on top of AWT, you can create a Swing project and use AWT code in it. Only create an AWT project if you're absolutely sure you won't need to use any Swing code in your program.

What project type do you choose if you want to use Ant to build your project, but also want to use the AWT or Swing frameworks? Use an Ant project, and add the AWT and Swing frameworks in your source code.

```
import java.awt.*;  
import javax.swing.*;
```

Xcode 2.2 added two Java project types: signed applets and Web Start applications. Signed applets are applets that contain a digital signature. Web Start is a technology that lets users easily download and launch Java applications from the Internet.

## Kernel Extension Projects

Apple's documentation discourages you from writing kernel extensions so you might be surprised that Xcode includes a project template for kernel extensions. Kernel extension programming is low-level operating system programming with the potential to wreck your computer so be careful if you choose to write a kernel extension. You're more likely to create an IO Kit Driver project, which you use to write device drivers for computer peripherals like printers, joysticks, and disk drives.

## Standard Apple Plug-In Projects

Use the standard Apple plug-in projects to write plug-ins for Apple programs. You can write plug-ins for Address Book, Core Image, Installer, Interface Builder, Sherlock, System Preferences, and Xcode.

You can also write screen savers, metadata importers, and sync schemas. Metadata importers extract metadata from files. Sync schemas synchronize data between your program and other applications. Core Image and Installer plug-ins, metadata importers, and sync schemas are not available in versions of Xcode prior to Xcode 2.0.

## Static Library Projects

Static libraries are pieces of code your application links against when compiling your code. You can create static libraries using the Carbon and Cocoa frameworks. The BSD static library project lets you write a static library using the BSD Unix APIs. If you're writing a static library you want to use on Linux or Unix operating systems, create a BSD static library project.

## Creating Your Own Project Templates

Xcode has a lot of project types to choose from, but you may need a project type that Xcode doesn't supply. If you write a lot of OpenGL programs, you would like to have an OpenGL application project that includes the OpenGL framework so you don't have to add the framework every time you create a project. When Xcode's project types don't fit your needs, create a project template.

A project template is just an Xcode project. When you create a new Xcode project using your project template, Xcode creates a project that contains everything in the project template. To create a project template:

- 1) Go to `/Library/Application Support/Apple/Developer Tools/Project Templates`. This folder contains all the Xcode project templates.
- 2) Select the project type that most closely matches the template you want to create. Make a copy of that project type's folder. This copy is your template folder.
- 3) The project folder's name is what appears in the Project Assistant window. Change the name of your template folder.
- 4) Open the project file that resides in your template folder.
- 5) Make the changes you need to make to the project. Add frameworks and files. Add code to a source code file. Do whatever you have to do to make the project a suitable template.

Now when you create a new project, you should see your project template as one of the project choices in the Project Assistant window. There is one thing missing. When you select your project template, the description of the project you copied appears. How do you change the description?

The description appears in the project's `TemplateInfo.plist` property list file. This file resides in the project file's bundle. Control-click the project file in the Finder to open a contextual menu. Choose Show Package Contents from the menu to open a Finder window that shows the bundle's contents. You should see the `TemplateInfo.plist` file in the newly opened window.

Double-clicking the `TemplateInfo.plist` file opens it in the Property List Editor application. Click the disclosure triangle next to Root. You should see a property named Description. The Description property's contents is what you will see when you select the project from the project list when you create a new project. Double-click the Value column for the Description property and type the project's description.

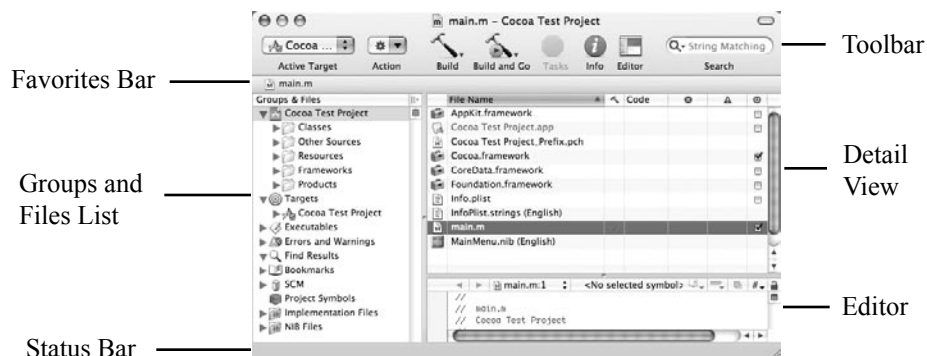
## The Project Window

The project window, shown in Figure 1.2, is your project's home when working in Xcode. It has the following components:

- Toolbar
- Favorites bar
- Groups and Files list
- Detail view
- Status bar
- Editor

I'm going to cover the toolbar and the Groups and Files list shortly. The favorites bar is initially invisible. Choose View > Show Favorites Bar to make the favorites bar visible. Use the favorites bar to store items you want to quickly access, such as source code files and documentation pages. To add an item to the favorites bar, select the item and drag it to the favorites bar.

The status bar reveals the progress of lengthy tasks, such as building your project. The editor is initially invisible. Drag the splitter bar at the bottom of the window to show the editor. Clicking the Editor button in the toolbar replaces the detail view with the editor. If you prefer to edit your source code in a separate window, leave the editor in the project window invisible. The point of showing the editor in the project window is to reduce window clutter by having only one window open.



**Figure 1.2**

Project window.



## Toolbar

At the top of the project window is the toolbar, which provides you easy access to tasks you perform most. Xcode places the following items on the toolbar by default:

- The Active Target menu lets you select which of your project's targets is the current one.
- The Action menu contains frequently used commands. What appears in the menu depends on what you select in the Groups and Files list.
- The Build Active Target button builds your project if you click it. If you hold the button down, a menu opens, giving you the option of cleaning your target. Cleaning the target removes all the object code, forcing you to recompile all your source code files.
- The Build and Go button builds your project and runs the program if you click it. If you hold the button down, a menu opens. From it you can choose to build and debug your program, run your program without building it, and debug your program without building it.
- The Tasks button will stop any programs you have executing in Xcode.
- The Info button opens the selected item's information panel. You will use this button to tweak compiler settings, which is a topic I will cover later in the chapter.
- The Editor button replaces the project window's detail view with an editor so you can edit your source code files. If you want to have only one window open, clicking the editor button gives you more room for editing. Click the Editor button a second time to restore the detail view.
- The Search field lets you filter listings in the project. If you wanted to see only header files in the project window, you would type `.h` in the search field.

If those items aren't what you use most, you can customize the toolbar to suit your needs. To customize the toolbar, choose View > Customize Toolbar. A window like Figure 1.3 will open. To add an icon to the toolbar, drag it to the toolbar. To remove an item from the toolbar, drag it off the toolbar. To rearrange items in the toolbar, drag the icon where you want it to appear.

You can also choose how the items on the toolbar appear. You can show the icon only, text only, or an icon with text. You can also make the icons smaller by selecting the Use Small Size checkbox. Click the Done button when you're finished tinkering with the toolbar.

## Groups and Files List

The Groups and Files list shows the contents of your project. Xcode projects have many files so Xcode places the files in groups. Control-clicking an item in the Groups and Files list opens a contextual menu. The contents of the contextual menu depend on the item you click, but the tasks you can perform from the contextual menu include adding files to a project, adding targets to a project, adding build phases to a target, compiling a single file, and building a target.



**Figure 1.3**

Customize toolbar window.

## Project Name

The first entry in the Groups and Files list is the name of your project. If you select it, the detail view displays all the files associated with your project: source files, frameworks, property lists, and executable files. Figure 1.4 shows what the detail view looks like when you select the project name from the Groups and Files list.

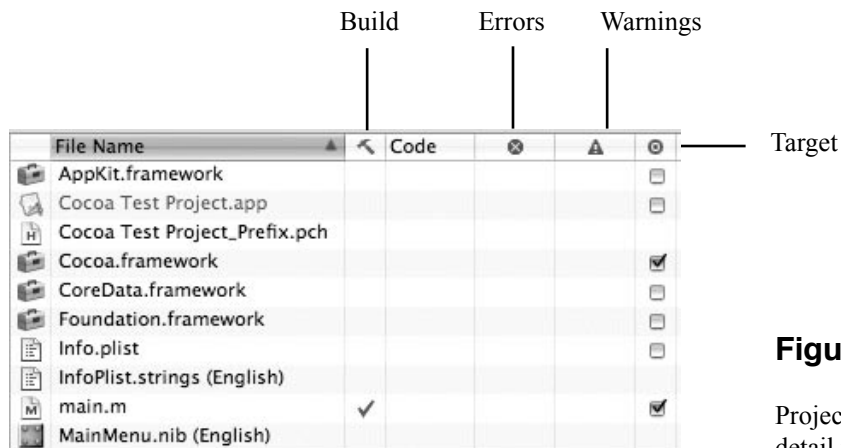
The detail view displays seven columns of information about each file in your project.

- An icon representing the type of file, such as framework, executable file, header file, or property list.
- The file's name.
- Does the file need to be built? If the file has a check in this column, it needs to be compiled.
- For source code files the Code column tells you the amount of code in the file. The Code column will be blank until you compile your project.
- The Errors column tells you how many compiler errors appeared in this file. This column applies only to source code files. You want the Errors column to be blank, which means your source code files have no errors.
- The Warnings column displays the number of compiler warnings for this file. You want this column to be blank as well as the Errors column.
- The Target column has a checkbox for your source code files and frameworks. If the checkbox is selected, the file is part of the active target.

Double-clicking a source code file or a property list file opens the file in a separate window. Double-clicking a nib file opens it in Interface Builder. Double-clicking a framework opens a window for it in the Finder. If you don't want to open separate windows, click the Editor button in the project window toolbar. It will add an editor pane to the project window so you can edit source code. Selecting a source code file or property list file from the project window will open the file in the editor pane. Xcode doesn't make the project windows very large. You will want to resize it if you're going to edit source code in the project window.

Xcode sorts the files alphabetically by default. By clicking one of the column headings, Xcode will sort the files by the column you clicked.

If you have a large project, you may have a hard time finding the files you're interested in. By clicking the disclosure triangle next to the project name in the Groups and Files list, Xcode displays folders for each group in your project. The initial group of folders a project has depends on the project you make, and you can add your own groups by choosing File > New Group. Selecting a folder will make that folder's files appear in the detail view.



**Figure 1.4**

Project window detail view.

## Targets

As the name implies, the Targets section lists your project’s targets. What is a target? A *target* is a set of instructions to build a final product from the files in your project. Examples of final products are applications, libraries, and frameworks. When you create an application project, Xcode automatically creates a target for building an executable file.

Click the disclosure triangle next to the Targets entry to see your project’s targets. Selecting a target fills the detail view with the files and frameworks that are part of the target.

Xcode projects initially have one target. The section “Adding Targets” explains how to add targets to a project. If you have multiple targets in your project, use the Active Target pop-up menu to choose the active target. The section “Working with Targets” provides more information on targets.

## Executables

The Executables group lets you tinker with the settings for the executable files your project creates. Obviously this group is important only for projects that create executable files, like application projects. Selecting Executables makes all the executable files for your project appear in the project window. Most of the time you will have only one executable file, but if you add targets to your project, there’s the possibility of multiple executable files.

Selecting an executable file in the project window and clicking the Info button opens the executable file’s information panel. The panel has four tabs.

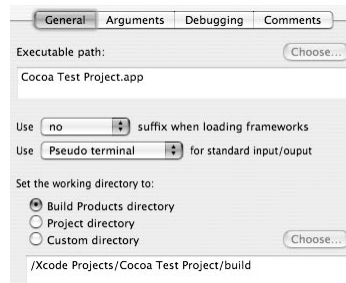
- General
- Arguments
- Debugging
- Comments

Click the Comments tab to add notes about the executable file’s settings.

## General Panel

The general panel, shown in Figure 1.5, is where you set the following information about the executable file:

- The path to the executable file.
- The type of frameworks to load when running your program.
- The program to use for standard input and output.
- The working directory.



**Figure 1.5**

Executable general panel.

For bundled applications the path is `ExecutableName.app`. Most Mac OS X applications are bundled. They appear in the Finder as a single executable file, but beneath the surface lays a web of directories. Inside this web is the actual executable file. Because bundles are complex to navigate manually, Xcode does not let you modify the path for bundled applications. If your application is not bundled, you can click the Choose button to set a path.

Use the Use suffix when loading frameworks pop-up menu to tell Xcode what type of framework to load. Xcode can load one of three versions of a framework.

- The standard variant is the default option and it runs the normal runtime library. This variant will be what you want most of the time.
- The debug variant provides additional debugging information while your program runs.
- The profile variant provides additional information for profiling your application to see where it spends the most time. Use the profile variant if you're using `gprof` to profile your program.

If you're writing a program without a GUI, you will appreciate the pop-up menu that lets you choose the program to use to display standard input and output. The program options are:

- Pseudo terminal, which is the default option. When you use pseudo terminal, command-line programs run in Xcode's run log the way they would run from the command line.
- System console. When you use system console your program's output appears in the system log instead of Xcode's run log. The advantage of using the system console is your program's output is saved to a log file for you to look at later. I recommend not using system console if your program uses standard input to get information from the user.
- Pipe is the option to use if you're going to run your program from another computer. Pipe lets you transfer input and output between the two computers.

Last comes the radio button group to set your program's working directory, the initial directory where your program looks for files. Most Mac OS X programs do not deal with working directories. By default the working directory your product directory, which is the build folder in your project directory. You can choose to go with the project folder, or you can select an entirely different directory as your working directory.

## Arguments Panel

Use the arguments panel to set the runtime arguments and environment variables your program uses. Tool projects are the most likely to use runtime arguments. To add a launch argument, click the + button and enter the argument name. To temporarily disable an argument, deselect the checkbox next to the argument. To remove an argument, select it from the list and click the minus button.

An *environment variable* is an operating system global variable that applications can use to send data to and receive data from other applications. Environment variables allow programs to reconfigure themselves while they're running. Unix programs use environment variables more than Mac programs.

To add an environment variable, click the + button. Type in a variable name and give it a value. To temporarily disable an environment variable, deselect the checkbox next to the variable. To remove an environment variable, select it and click the minus button.

If you want to experiment with environment variables, add one by clicking the + button. Give the variable the name `MallocScribbling` and the value 1. Now when you run your program, the operating system will take the memory your program frees and fill the freed memory with garbage values. Normally the memory would remain unchanged when you free it. Setting the `MallocScribbling` environment variable lets you discover if you're accessing memory you've already freed; your program will crash if you access freed memory.

## Debugging Panel

The debugging panel is where you set options related to debugging the executable file. I cover debugging next chapter, in which I cover the debugging panel in more detail.

## Errors and Warnings

The Errors and Warnings group lists any compiler errors or warnings in your project. This group will be empty until you compile the files in your project. Hopefully the group will remain empty after compiling. For each error and warning in your program, Xcode reports the following information:

- Was it an error or a warning? Errors prevent the program from being built. Warnings alert you to possible problems in your program.
- The error or warning message.
- The file where the error or warning occurred and the line number.

Double-clicking a listing in the Errors and Warnings group takes you to the line of code where the error or warning occurred. From there you can fix the problem.

## Find Results

The Find Results group lists the results of searches you made for this project. It's helpful if you search for a particular term in your project, then want to look for it again later. Xcode stores the results temporarily; when you close your project, you lose the search results.

## Bookmarks

The Bookmarks group lists all the bookmarks you've set. Bookmarks let you quickly reach a line in one of your source code files. To add a bookmark, move the cursor to a line in one of your files, then choose File > Add to Bookmarks. Xcode opens a dialog box to name the bookmark. By default Xcode lists the source code file and line number, but you can give the bookmark any name you want. If you click the Bookmarks entry in the Groups and Files list, you will see the bookmark you created.

## SCM

The software configuration management (SCM) group works with version control, which tracks the changes you make to files. Version control works by storing all your files in a repository. You check files out of the repository, make the changes you want to make to them, and check the files back in. The SCM group lists the files you have modified but haven't checked back in. If you're new to Xcode, the SCM group will be blank because you haven't set up version control yet. Version control is such a large subject, I devote an entire chapter to it. Refer to Chapter 11 for more information on using version control with Xcode.

## Project Symbols

Selecting Project Symbols from the Groups and Files list provides lots of information. The project symbols group shows every class you wrote, the data members of the classes, every function you wrote, every constant you defined, every enumerated data type you created, and every macro you defined.

Figure 1.6 shows a sample listing of project symbols. The listing has three columns. The first column is the name of the symbol. There's a symbol for every class, data member, function, constant, data type, and macro in your program. The second column describes the type of symbol. Table 1.1 provides a list of common symbols. The final column tells you the file and the line number where the symbol appears. Double-clicking a symbol opens up a new editor window and takes you to the symbol.

**Table 1.1 Common Project Symbol Types**

Symbol Type	Description
Class	A class name. You will have one of these symbols for each class you create in an object-oriented language like C++, Java, or Objective C.
Class Method	Static functions in a class. Normally when you declare objects of a particular class, each object has an instance of each member function in the class. Static functions have one instance for the class. All objects of the class share the one instance of the static function.
Constant	If you use enumerated data types, each value in the data type has a Constant symbol.
Enum	The name of an enumerated data type.
Function	Functions that are not member functions of a class. C functions, C++ constructors, and C++ destructors have Function symbols.
Instance Method	Non-static member functions of a class. If you use an object-oriented programming language, most of the functions you write will have Instance Method symbols.
Instance Variable	The data members of a class.
Macro	Macros that you create with the <code>#define</code> statement. C programs use macros to define constants.
Type	If you use the <code>typedef</code> statement in C to define new data types, the new data types appear in the symbol list as Type symbols.
Variable	Variables declared outside of a class. If you use constants for special values to avoid hard-coding numbers, the constants appear in the symbol list as Variable symbols.

Symbol	Kind	Location
M AllocateLevelMap	Instance Method	GameLevel.h:54
M AllocateLevelMap	Instance Method	GameLevel.cpp:115
M AnimatePlayer	Instance Method	GameApp.h:58
M AnimatePlayer	Instance Method	GameApp.cpp:256
M CleanUpApp	Instance Method	GameApp.h:38
M CleanUpApp	Instance Method	GameApp.cpp:94
M Create	Instance Method	GameTexture.h:64
M Create	Instance Method	GameTexture.cpp:182
M DeleteLevelMap	Instance Method	GameLevel.h:55
M DeleteLevelMap	Instance Method	GameLevel.cpp:121
M DetermineUserAction	Instance Method	GameApp.h:56
M DetermineUserAction	Instance Method	GameApp.cpp:206
V done	Instance Variable	GameApp.h:27
M Draw	Instance Method	GameTexture.cpp:200
M Draw	Instance Method	GameTexture.h:67

**Figure 1.6**

Project symbols listing.

## Smart Groups

Smart groups do not have the name Smart Groups. They're folders that group files automatically using rules you specify. Xcode projects come with two smart groups: Implementation Files and NIB Files. The Implementation Files folder contains the AppleScript, C, C++, Java, Objective C, Objective C++, and shell script files in your project. The NIB Files folder contains your project's nib files, files with the extension `.nib`.

If your project doesn't contain many files, you won't need smart groups, but for large projects, creating your own smart groups makes locating files easier. To create a smart group, choose **File > New Smart Group**. There are two types of smart groups: simple filter and simple regular expression. It doesn't matter which one you choose right now. You can change the type of smart group later.

The smart group you create initially has the name Simple Filter Smart Group or Simple Regular Expression Smart Group. You're going to want to change the name. Select the smart group you created from the project window and click the Info button. The group's information panel opens, as you can see in Figure 1.7.

From the information panel you can choose the name and folder icon for the group. You can also choose the starting point for grouping the files. By default it's the project folder, which means Xcode looks at all files in the project. You can limit the search to a specific folder inside the project if you wish.

In the Using Pattern text field, you type in the pattern Xcode uses to determine what files appear in the group. There's also a radio button determining whether you want to use a wildcard pattern or regular expression. That's why I said it didn't matter which type of smart group you created initially; you can change it here.

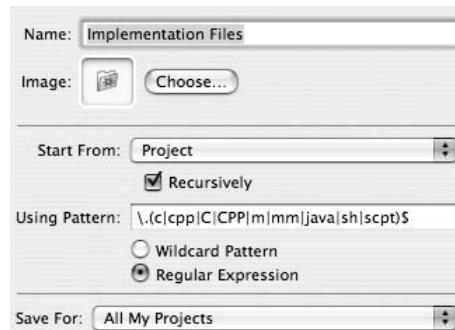
Wildcard patterns are easier to use. Type in the filter you want to use, and you're done. All the files that match the filter will appear in the group. If you wanted to create a smart group for header files, you would type in the pattern `*.h`. Every file with the extension `.h`, the extension that header files use, will appear in the group.

Regular expressions give you more power. Enter any Unix expression, and Xcode places files that satisfy the expression in the group. To see an example of a smart group that uses regular expressions, select the Implementation Files group and click the Info button. The Implementation Files group uses the following expression:

```
\.(c|cpp|C|CPP|m|mm|java|sh|sct)$
```

This expression says that any file that contains the characters `.c`, `.cpp`, `.C`, `.CPP`, `.m`, `.mm`, `.java`, `.sh`, and `.sct` appears in the group. These characters just happen to be the default file extensions for C, C++, Objective C, Objective C++, Java, shell script, and AppleScript files.

Use the Save For pop-up menu to determine whether you want the group you created to appear in this project only or in all Xcode projects you create.



**Figure 1.7**

Smart group  
information panel.

## Adding Files and Frameworks to Your Project

Xcode includes a source code file when you create a new project, but unless you're writing a very simple program, you're going to be adding files to your project. You can create new files or add existing files to your project.

### Adding New Source Code Files to the Project

The most common case of adding files to a project is creating new source code files. To create a new file, choose File > New File. Creating a new file consists of two steps.

- Choosing a file type.
- Naming the file.

Enterprise Java Beans and servlet files have a third step. The third step involves setting properties for the file you're creating.

### Choosing a File Type

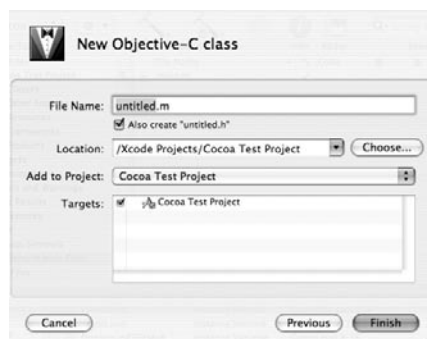
When you create a new file by choosing File > New File, the New File Assistant dialog box opens. The dialog box shows a list of files you can create. Selecting a file type from the list displays a description of the file type.

Select the type of file you want to create from the list. Click the Next button to move on to naming the file. If you choose to create a J2EE Enterprise Java Beans file, you must choose the type of bean you want to create before you can name the file.

The file list contains nine types of Cocoa files: four Java files and five Objective C files. You can create Objective C test case files, plain class files and subclasses of `NSDocument`, `NSView`, and `NSWindow`. If you're not sure what type of Cocoa class you should create, choose a Cocoa class file instead of one of the subclass files.

### Naming the File

After selecting the type of file you want to create, click the Next button, which will take you to the second part of the file creation process. Figure 1.8 shows what the dialog box looks like. Name your file. Notice that Xcode automatically includes the appropriate ending depending on the type of file you create. A C file will have the extension `.c`.



**Figure 1.8**

Xcode's New File Assistant.



If you created a C, C++, or Objective C file, there will be a checkbox with the caption Also create FileName.h, where *FileName* is the name of your file. C, C++, and Objective C programs normally have one header file for each source code file. The header file defines data structures and declares functions. The source code file contains the source code for the functions you declared in the header file. In most cases you want to keep the checkbox selected. Because Xcode creates header files when it creates a source code file, you shouldn't need to explicitly create a header file often.

After naming the file tell Xcode where you want the file to reside on your hard disk. By default it will be in the same folder where your project is. You also have the option to determine which project you want to add the file to. By default it's the current project, but you can choose another project you have open in Xcode or choose not to add it to any project.

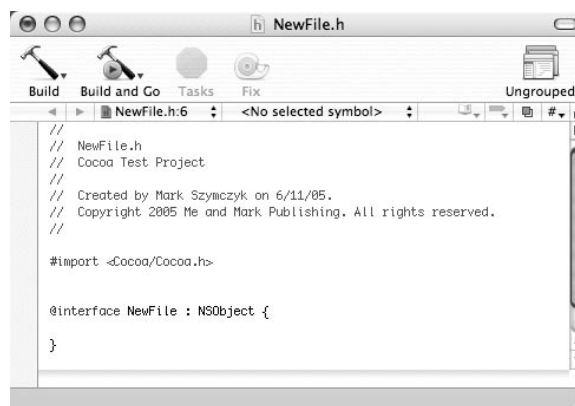
Click the Finish button to create the file. Enterprise Java Beans and servlet files have another dialog box of properties to set before you can click the Finish button. Figure 1.9 shows an example of the file Xcode creates. It includes a comment containing the following information:

- The file name.
- The project name.
- A notice saying who created the program and when.
- A copyright notice.

### Creating Your Own File Templates

Xcode has many file types to choose from, but Xcode's file types may not suit your needs. You may write code in a language that Xcode doesn't include in its new file list. You may want header files from additional frameworks to be included when you create a new file. Create a file template when Xcode's file templates aren't right for your programs. To create a file template:

- 1) Choose File > New Empty File to create an empty file.
- 2) If you want to include a header file with the implementation file, create a header file as well.
- 3) Put what you want in the file.
- 4) Save the file. It should have the extension you want files of this type to automatically have. If you were creating an Objective C++ file template, you would give the file the extension `.mm`.
- 5) Move the file and any header file into its own folder. The folder should have the extension `.pbfiletemplate`.



**Figure 1.9**

A new file that Xcode creates.

Before you can add your template to Xcode's file template list, you must add a property list file named `TemplateInfo.plist`. To create the property list file:

- 1) Launch the Property List Editor program.
- 2) Click the New Root button.
- 3) Click the disclosure triangle next to Root.
- 4) Click the New Child button twice.
- 5) Give one child the name `Description` and the other child the name `MainTemplateFile`.
- 6) The value of `Description` is the description you want to appear in the New File Assistant window.
- 7) The value of `MainTemplateFile` is the name of the file template you created.

If you included a header file, click the New Child button a third time. The property name should be `CounterpartTemplateFile`, and the value should be the name of the header file you created.

After creating the `TemplateInfo.plist` file, move it to the folder containing your file template. Move the folder to `/Library/Application Support/Apple/Developer Tools/File Templates`. This folder contains all the Xcode file templates. Now you can create these files in Xcode.

## Fixing the Copyright Notice

If you look at the copyright notice of the files you create, it will probably contain the text `MyCompanyName`. Unless you happen to work for a company called `MyCompanyName`, you want Xcode to use your company name for the copyright notice. To change the company name you must edit Xcode's preferences file. The preferences file resides in the folder `/Username/Library/Preferences` and has the name `com.apple.Xcode.plist`. Double-click the preferences file to open it in the Property List Editor program. Select Root and click the disclosure triangle next to Root to see all the preferences.

The text for the button in the upper left corner should change to New Child. Click the New Child button to create a new preference. Type the name `PBXCustomTemplateMacroDefinitions`. Select this preference. Its class should be String. Click on the word String to open a menu; change the class to Dictionary. Changing the class adds a disclosure triangle to the left of the `PBXCustomTemplateMacroDefinitions` property.

Click the disclosure triangle next to `PBXCustomTemplateMacroDefinitions`. Click the New Child button to add an item to the dictionary. The item's initial name is New Item. Change it to `ORGANIZATIONNAME`; make it all uppercase letters. The item has class String, which is what you want. The item's value is currently empty. Double-click the value to type in your company name. If you don't work for a company, type your name. Figure 1.10 shows you what you need to type. Save the file. Quit Xcode and relaunch it to have the changes you made take effect. If you don't quit Xcode, it uses the old settings, which means the files you create will say `MyCompanyName` instead of your company's name.

Property List	Class	Value
<code>PBXCopySourceCodeAsPlainText</code>	Boolean	Yes
▼ <code>PBXCustomTemplateMacroDefinitions</code>	Dictionary	1 key/value pair
<code>ORGANIZATIONNAME</code>	String	Your Company Name

**Figure 1.10**

Adding your company name to Xcode's property list so your company name appears in source code files that Xcode creates.

## Adding Files You've Already Created

Having to write every line of code from scratch for every project would get tiresome quickly. Adding files you created for previous projects makes developing new programs easier. To add files you've already created to a project, choose **Project > Add to Project**. A dialog box opens that lets you find the files you want to add. After selecting files to add, a sheet like Figure 1.11 will open. Click the **Add** button to add the files.

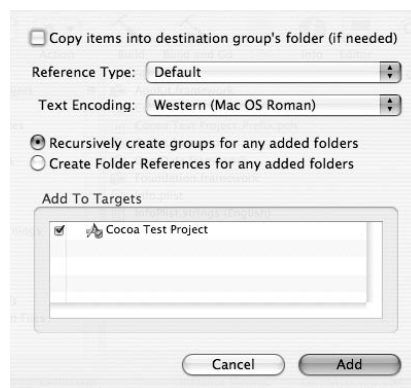
Selecting the **Copy items into group's destination folder (if needed)** checkbox tells Xcode to copy the files you added into your project's folder if the files aren't already there. The **Reference Type** determines how Xcode stores the file's location. There are four options.

- By group, which means Xcode uses the file's group in the Groups and Files list to store the file's location.
- By project, which means Xcode uses the project's folder to store the file's location.
- Absolute path, which means Xcode uses the file's path on your computer to store the file's location.
- By build product, which means Xcode uses the folder where it places build products, such as executable files and libraries, to store the file's location.

The default reference type for files is by group. The reference type options matter if you're going to share your projects with other people. If you referred to your files according to the absolute path on your computer, chances are high that other people using your project would have a different path to the files. In this case Xcode would be unable to find the files and the project would not compile.

If you're adding an entire folder of files, you have two options: recursively create groups for any added folders and create folder references for any added folders. When you recursively create groups, Xcode creates a group in the Groups and Files list for each folder you add. Recursively create groups when you need to access the individual files of the folder in Xcode. If you're adding a folder of source code files, you want to recursively create the group because you want to be able to examine and edit the source code files.

When you create a folder reference, Xcode adds the folder to the project, but not the files in the folder. Create a folder reference when you don't need to access the individual files in Xcode. If you're writing a game and you have a folder named **Sound Effects** that holds the game's sound effects, create a folder reference when adding the **Sound Effects** folder to your project. You're not going to be doing anything with the sound files in Xcode. All you want to do is copy the folder to the application bundle when you build the project, and creating a folder reference lets you accomplish this task.



**Figure 1.11**

Add File dialog.

## Source Trees

There can be a fifth Reference Type for files you add to a project: by source tree. A *source tree* is a root path to place files that you want multiple people to work on. Source trees allow your project's files to remain independent of the project folder. The point of using source trees is to allow multiple people to work on a project. You can transfer the project to other people's computers without messing up the project's file references. To create a source tree:

- 1) Choose Xcode > Preferences to open Xcode's preferences panel.
- 2) Select Source Trees from the preferences panel.
- 3) Click the + button to add a source tree.
- 4) Click the OK button.

There are three pieces of information you must supply to add a source tree.

- Setting Name is the name of the tree. Everyone who wants to use a source tree must give it the same setting name on their Macs.
- Display Name is the name Xcode shows for the source tree.
- Path is the location of the source tree on your hard disk.

Each person who wants to use a source tree can store it wherever they want on their hard disk. As long as everyone uses the same name for the source tree, everyone can use it. If you want other people to work on your projects, copy the files from the location of the source tree on your Mac to the location of the source tree on their Macs.

## Adding Frameworks

If you're writing a program consisting mostly of GUI code, you can get away with using just the Carbon or Cocoa frameworks. Many popular Apple technologies, such as QuickTime, OpenGL, and Core Audio, require you to add a framework to your project. Adding a framework to your project is identical to adding previously created files. Choose Project > Add to Project. You can find most of Apple's frameworks in `/System/Library/Frameworks`. After selecting the frameworks you want to add, the sheet shown in Figure 1.11 opens. Click the Add button to add the framework to your project.

## Editing Source Code

When you develop software you spend a lot of time writing and editing source code. This section covers Xcode's features for editing source code.

### The Editor Window

Double-clicking a source code file in the project window opens an editor window. Figure 1.12 shows a typical editor window, which contains five parts.

- Toolbar
- Status bar
- Navigation bar
- Editor
- Gutter

## Toolbar

The toolbar, which runs along the top of the editor window, contains commonly used commands. The toolbar in Figure 1.12 contains the default toolbar items. The buttons on the left side of the toolbar have nothing to do with editing source code. The first two buttons deal with compiling and running your program. The Fix button works with debugging, and the Tasks button stops tasks like compiling, running, and debugging your program.

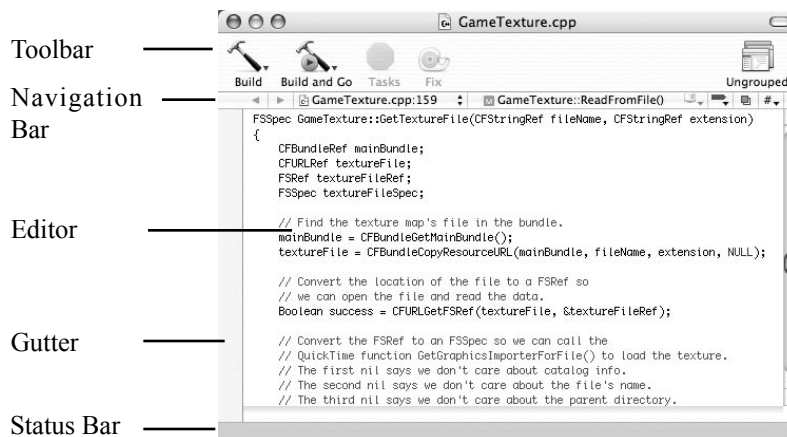
The Grouped button (Editing Mode button in some versions of Xcode) is the most interesting toolbar button for source code editing. The initial editing mode is single window. Single window editing mode means there's only one editor window open at a time. When you open another file Xcode places the new file's contents in the window. Clicking the Grouped button changes the editing mode to multiple window, and the button name changes to Ungrouped. When you use multiple window editing mode, each source code file opens in its own window.

## Customizing the Toolbar

Choose View > Customize Toolbar to customize the editor window's toolbar. Make sure an editor window is the current window before choosing View > Customize Toolbar. A sheet will open in front of the window. Customizing the editor window toolbar is similar to customizing the project window toolbar; drag the icons you want to add from the customization sheet to the toolbar. Click the Done button to finish customizing.

## Status Bar

The status bar doesn't do much when you're editing source code. It shows you the status of Xcode activities like building and debugging programs. The status bar can be in one of two places, depending on your version of Xcode. It can be below the toolbar or at the bottom of the window.



**Figure 1.12**

Xcode's editor window.

## Navigation Bar

The navigation bar, shown in Figure 1.13, contains controls to quickly reach areas of the current file and quickly change files. It has four different areas. On the left edge of the bar are back and forward buttons, which work similarly to an Internet browser's back and forward buttons.

To the right of the back and forward buttons is the recently viewed files list. The list displays the file you're viewing and the line number. The pop-up menu next to the file name and line number shows the list of recently viewed files. The recently viewed files list helps when you edit a file, edit a few more files, and want to go back to the first file you were editing.

Next to the recently viewed files list is the function list. It shows the function you're at in the file. There's a pop-up menu containing all the functions in the file. Use the function list to reach a function in the file quickly.

At the right edge of the navigation bar are four buttons. Clicking the leftmost button lists the bookmarks you've set in the file. Clicking the second button lists the breakpoints you've set in the file. The third button is the Go To Counterpart button, which works with C-based languages. If the current file is an implementation file, clicking the Go To Counterpart button opens the header file for the current file. If the current file is a header file, clicking the Go To Counterpart button opens the implementation file. Clicking the fourth button lists all the files the current file includes. Selecting a file from the list opens that file.

Some versions of Xcode have a fifth button. Use this button to lock and unlock files. Locking a file prohibits anyone from editing the file.

## Editor and Gutter

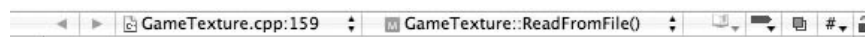
The editor is where you edit source code. You've used word processors and text editors before. Xcode's editor isn't radically different from editors you've previously used. The gutter runs along the left edge of the window. Its main use is for setting breakpoints, which I will discuss next chapter.

## Code Completion

Code completion tells Xcode to finish the names of your program's functions and variable names, saving you from having to type the whole name.

## Using the Completion List

To perform code completion, start typing the function or variable name and press the Esc key. A pop-up menu known as a *completion list* will open. The completion list contains all the possible matches. Select an entry from the completion list and press the Return key to complete the code. If you don't want to complete the code, press the Esc key to close the completion list.



**Figure 1.13**

The editor window's navigation bar.

## Cycling Through Completion Matches

An alternative method of code completion is to cycle through the possible matches instead of opening a completion list. Pressing Control-period displays the next match in the editor. Keep pressing Control-period until your desired match appears.

## Customizing Code Completion

For code completion to work, you must have the Enable Indexing for all projects checkbox selected in Xcode's preferences panel, shown in Figure 1.14. Xcode creates a project index that keeps track of all the project symbols, such as variable names, function names, and constants. Xcode uses this index for many things, one of which is code completion. If you turn off indexing, you disable code completion as well.

Xcode has four checkboxes to customize how code completion works. When you select the Indicate when completions are available checkbox, Xcode underlines the text you're typing if the text has completions available.

Selecting the Show arguments in pop-up list checkbox tells Xcode to add arguments for each function that appears in the completion list. If you don't select the checkbox, the completion list shows the function name only. Adding arguments helps when you have functions with the same name but different arguments. Without the arguments you would have no way of knowing the function you were completing.

When you select the Insert argument placeholders for completions checkbox, Xcode inserts placeholders for the function's arguments when it completes a function. If you don't select the checkbox, Xcode completes the function name only. Here's how Xcode would complete the OpenGL function `glVertex3f()` with the function name only:

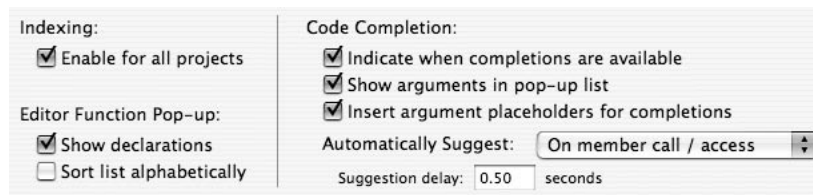
```
glVertex3f
```

And here's how Xcode would complete the `glVertex3f()` function with argument placeholders:

```
glVertex3f(<#GLfloat x#>,<#GLfloat y#>,<#GLfloat z#>)
```

If you tell Xcode to insert argument placeholders, press Control-slash to move to the next argument in the function. The Control-slash combination makes it easy to replace the argument placeholders with your own arguments.

Use the Automatically suggest pop-up menu to open the completion list automatically. Xcode can automatically open the completion list in two ways: open the list all the time or open the list when accessing your classes' members. You specify the amount of time that must pass before Xcode opens the completion list.



**Figure 1.14**

Code Sense preferences panel.

## Getting Information About Functions and Variables

Command-double-clicking a symbol in your source takes you to that symbol's declaration point. For class variables Xcode opens the header file where you declared the variable. For functions you wrote, Xcode takes you to the function. For functions in external frameworks, Xcode opens the header file and shows you the declaration. The header files in Apple's frameworks contain documentation about the framework's functions. Command-double-clicking doesn't work for local variables.

## Customizing Source Code Editing

To customize your experience editing source code, open the Xcode preferences panel by choosing Xcode > Preferences. The preferences areas of interest are the following:

- Text Editing
- Fonts and Colors
- Indentation
- Key Bindings

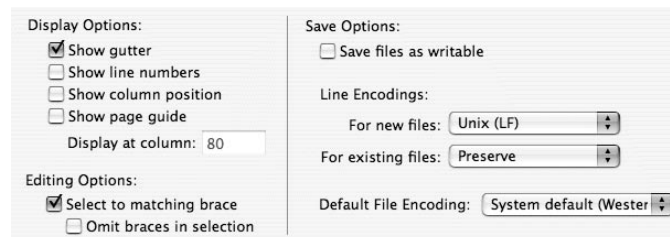
### Text Editing Preferences

The text editing preferences, shown in Figure 1.15, is where you set miscellaneous text editing options. Some options you can set include the following:

- What pressing the Return key to end a line of code generates: a carriage return or a line feed. Mac OS X, Unix, and Windows end lines differently. Mac OS X generates a carriage return. Unix generates a line feed. Windows generates a carriage return and a line feed.
- Showing the gutter on the left side of the editor window.
- Showing line numbers in the editor.

### Fonts and Colors Preferences

Xcode gives different areas of your code their own text color to make reading the code easier. Comments have green text. Numbers have blue text. Strings and language keywords (words used by the language such as `if`, `while`, `for`, and `int` in C) have red text. The syntax coloring preferences is where you set the text color, font, and point size for your code.



**Figure 1.15**

Text editing preferences.



## Indentation Preferences

Programmers use indentation to make the program's logic clearer. Compare the following code snippets:

```
for (short row = 0; row < rowCount; row++) {
for (short column = 0; column < columnCount; column++) {
if (IsRectangleDirty(row, column)) {
DrawTile(row, column);
}
}
}
```

```
for (short row = 0; row < rowCount; row++) {
    for (short column = 0; column < columnCount; column++) {
        if (IsRectangleDirty(row, column)) {
            DrawTile(row, column);
        }
    }
}
```

The second snippet is easier to read because of the code indentation. The indentation preferences let you turn on automatic code indenting, which Xcode calls syntax aware indenting. When you type a statement that calls for indentation, such as `if`, `for`, and `while` statements, and press the Return key, Xcode indents the new line for you. If you enter the statement without a leading brace, such as the following:

```
if (x == 0)
    y = 0;
else
    y = 1;
```

Xcode's indenting is smart enough to know an `if` statement without a brace has just one statement following it. When you press Return after typing the line `y = 0`, Xcode will line up the cursor with the `if` statement so the `if` and `else` lines start at the same column.

## Key Bindings Preferences

Use the key bindings preferences to set keyboard shortcuts for practically anything. You can set a keyboard equivalent for any menu item in the Xcode menu bar using the Menu Key Bindings tab. The Text Key Bindings tab lets you set keyboard equivalents for things like text editing, moving the cursor, and text formatting.



**Figure 1.16**

Class browser window.

Xcode comes with default key binding sets for Xcode, BBEdit, CodeWarrior, and MPW. If you're moving to Xcode from BBEdit, CodeWarrior, or MPW, the key binding sets will smooth the transition to Xcode. To create a custom set of key bindings, click the Duplicate button to create a copy of one of the default sets. Use the copy to bind keys.

## Using Xcode's Class Browser

For those of you using object-oriented languages like C++, Java, and Objective C, Xcode has a class browser to examine your classes' members. Choose Project > Show Class Browser to open the class browser, which you can see in Figure 1.16. The class browser has three areas.

- Class list
- Member list
- Editor

If no classes appear in the browser, you may need to index your project. To index your project, select the project name from the Groups and Files list and click the Info button. The project's information panel will open. Click the General tab. Click the Rebuild Code Sense Index button to create the index.

## Browsing Classes

The class list runs down the left side of the window. Classes you wrote appear in blue text. Classes defined in another framework, such as the built-in Cocoa classes, appear in black text. If a class has subclasses, that class will have a disclosure triangle next to it. Click the disclosure triangle to see the subclasses.

Selecting a class from the class list will do two things. First, Xcode fills the member list with the class's members. Second, Xcode opens the class's header file in the editor. Selecting a method from the member list takes you to the method's declaration in the header file.

## Customizing What the Class Browser Shows

You can customize what appears in the class browser by clicking the Configure Options button in the class browser toolbar. Doing so will open the class browser configuration window, shown in Figure 1.17. Looking at the figure, you can see two columns of controls. The left column deals with what appears in the class list, and the right column deals with what appears in the member list. Read the next two sections to learn what you can customize for the class and member lists.

At the top of the configuration window is a pop-up menu containing saved configurations. A saved configuration saves your settings so you don't have to configure the class browser every time you use it. The class browser comes with four saved configurations, and you can add configurations to the list. Click the Add button to create a saved configuration. Name the configuration, use the configuration window to modify the class browser settings, and click the OK button. The configuration you created now appears in the Option Set pop-up menu in the class browser window.



**Figure 1.17**

Class browser configuration window.

## Customizing What Appears in the Class List

In the Class List Display Settings section there's a radio button group asking you whether you want the classes in your project to appear in a hierarchical or a flat profile. In a hierarchical profile, classes with subclasses have a disclosure triangle, which you click to see the subclasses. In a flat profile the classes appear in alphabetical order.

Below the radio button group are three pop-up menus. The first pop-up menu lets you decide whether you want only classes from your project, only classes from external frameworks (Cocoa or Carbon for example), or both types of classes to appear in the class browser. The second pop-up menu lets you decide whether you want only classes, only protocols and interfaces, or both to appear in the browser. Protocols and interfaces are different names for the same thing. Objective C uses the term protocol, and Java uses the term interface. A *protocol* is a collection of method declarations. These declarations are not part of a class. Any class can implement the methods in the protocol.

The third pop-up menu applies only to Objective C programs. It lets you determine how to show Objective C categories. *Categories* let you extend existing classes by adding methods to them, which is something you can't do in C++ and Java. By using categories you can add methods to the built-in Cocoa classes. You can show categories as subclasses of the class you're extending, subclasses for root classes, or show them merged into the class itself.

## Customizing What Appears in the Member List

In the Member List Display Settings section there's a checkbox asking you if you want to show inherited members of a class. If you select this checkbox, the inherited members of the class appear in gray text in the member list. The non-inherited members appear in black text to distinguish them.

Below the checkbox are two pop-up menus. The first pop-up menu asks whether you want to show methods, data, or both in the member list. By default the class browser shows methods only.

The second pop-up menu asks whether you want to show the instances, show the class, or both. *Instances* are the class's data members and non-static member functions. If you choose class only, only static functions will appear in the member list. In Objective C code, only class methods will appear if you choose class only; no instance methods will appear in the member list. Because most functions in classes are non-static, you'll want to see instances in the class browser.

# Using Xcode's Modeling Tools

Apple added visual modeling tools in Xcode 2.0, the version that ships with Mac OS X 10.4. There are two modeling tools: the class modeling tool and the data modeling tool. The class modeling tool works with C++, Java, and Objective C programs. The data modeling tool works with Cocoa applications that use the Core Data framework.

## Modeling Classes

Xcode's class modeling tool shows the same kinds of information the class browser shows. You can examine your project's classes, view the members of each class, and see the inheritance relationships each class has. Why would you want to use the class modeling tool instead of the class browser? The class modeling tool has the following advantages:

- You have more control over what appears in a class model than what appears in the class browser.
- The class modeler draws a class diagram that lets people see your program's structure without having to look at source code.
- You can add comments that explain what each class does.
- You can track the changes you make to a class model in a version control system.

## Adding a Class Model to Your Project

To use the class modeler you must create a class model file to store your project's class models. There are two types of class models you can create: class models and quick models. Xcode adds class models to your project automatically. Class models also let you control what ends up in the model. Quick models add every eligible file to the model. Quick models are initially temporary. You must modify the quick model and save the changes for Xcode to add the quick model to your project.

Choose Design > Class Model > Quick Model to create a quick model file. If you want to add the quick model file to your project, modify the file and save it. You don't have to do much to modify the file. Looking at a class's member functions and data members in the class diagram modifies the file.

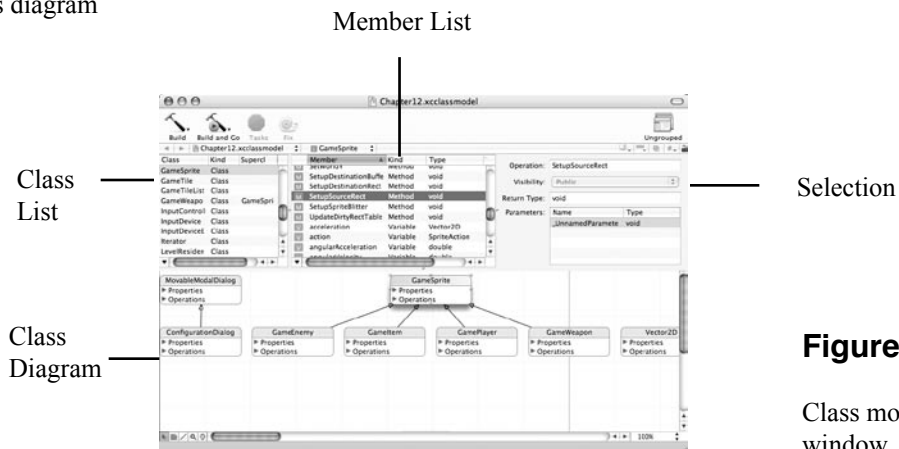
To create a class model:

- 1) Choose File > New File to create the file.
- 2) Select Class Model from the file list and click the Next button.
- 3) Name the file, select the targets you want to add the class model to, and click the Next button.
- 4) Add the file groups you want to be in the class model and click the Finish button. The classes in the groups you add are the classes that will appear in the class model. Click the Add All button to add all eligible files to the model.

## The Class Model Window

The class model window, shown in Figure 1.18, is where you examine your class models. The window has four sections.

- Class list
- Member list
- Selection area
- Class diagram



**Figure 1.18**

Class model window.

## Class List

The class list shows all the classes in the model. For each class you can see what superclasses the class has and the kind of class it is. Most classes are of kind Class, which means they're ordinary classes. Objective C programs have two additional kinds of classes: categories and protocols. A category is a collection of methods to add to an existing class. A protocol is a collection of method declarations. These declarations are not part of a class. Any class can implement the methods in the protocol. Java has protocols as well, but Java calls them interfaces.

If a class has a book icon in the far right column, that class has documentation you can look at. Cocoa and Java classes are the most likely to have documentation available.

## Member List

Selecting a class from the class list fills the member list with the class's members. The member list displays the following information for each member:

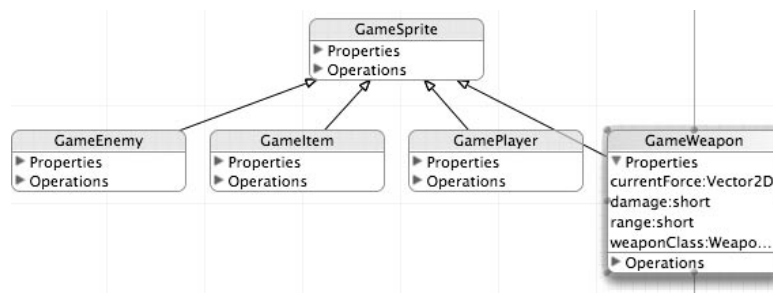
- An icon describing the type of member. Member functions have the letter M for method, and data members have the letter V for variable.
- The member name.
- The member kind: method, variable, or class.
- The member's data type. The data type for methods is the data type the method returns.
- The member's visibility: public, private, or protected.
- Whether or not the member has documentation. A book icon signifies that the member has documentation.

## Selection Area

Selecting a member from the member list fills the selection area with information about the member. If you select a method, the selection area tells you the name and data type of each argument the method takes.

## Class Diagram

The class diagram, shown in Figure 1.19, contains the classes in the model and the relationships between them. A solid line connecting two classes indicates inheritance, with the line going from the subclass to the superclass. A solid line connecting a class and a category indicates the class that the category is a category of. A dashed line indicates a protocol (or interface for Java programs) implementation.



**Figure 1.19**

Class diagram.

Each class in the diagram has three compartments: the class name, properties compartment, and operations compartment. Properties are the data members of the class, and operations are the methods. Initially Xcode shows the class name, properties compartment, and operations compartment, but leaves the class's properties and operations hidden. Click the disclosure triangles to show the properties and operations in their respective compartments. Choosing Design > Roll Up Compartments shows the class name only. Choosing Design > Roll Down Compartments restores the properties and operations compartments.

There are two ways to organize the classes in the diagram: hierarchically and force-directed. Hierarchical layout places parents at the top of the diagram with their children below. Force-directed layout places classes that other classes reference in the center of the graph. A force-directed graph takes longer to create. If you have a lot of classes in your class model, you should use hierarchical layout. Choose Design > Automatic Layout to choose the layout.

## Opening the Information Panel

Opening the information panel for a class model is slightly different than opening other information panels. Click the diagram without selecting anything. Choose File > Get Info to open the information panel. The information panel should look like Figure 1.20. If the information panel has only the Appearance tab, make sure you didn't select a class or a comment in the diagram before choosing File > Get Info.

## Customizing the Class Diagram

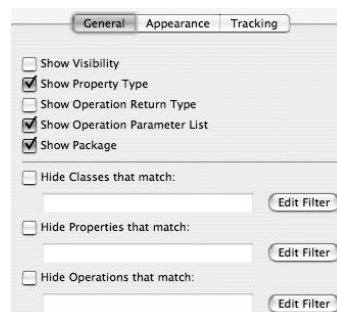
To customize the information that appears in the class diagram, open the class diagram's information panel and click the General tab. At the top of the panel are five checkboxes that control the class member information that appears in the diagram. Selecting the Show Visibility checkbox tells Xcode to show the visibility (public, private, or protected) of each property and operation.

Selecting the Show Property Type checkbox tells Xcode to show the data type of each data member. Selecting the Show Operation Return Type checkbox tells Xcode to show the return type of each class method.

Selecting the Show Operation Parameter List checkbox tells Xcode to show the parameters each method takes. Selecting the Show Package checkbox tells Xcode to show the package that a Java class belongs to.

## Adding Comments

Xcode class diagrams can have comments that provide an explanation of what a class does. Choose Design > Class Model > Add Comment to create a comment. Select the comment in the class diagram to enter text for the comment. To attach a comment to a class, use the line tool to draw a line from the class to the comment. The line tool is in the lower left corner of the class model window.



**Figure 1.20**

Class model  
information panel.

## Filtering Information from the Diagram

To filter information from the diagram, open the class model's information panel and click the General tab. There are three checkboxes at the bottom of the information panel. Use them to filter classes, methods, and data members from the diagram. Click the Edit Filter button to open the predicate builder, which lets you specify filter conditions.

Click the + button to add a predicate and click the minus button to remove a predicate. Use the left pop-up menu to specify what you're comparing. Use the right pop-up menu to specify the condition. Use the text field to specify the value. If you wanted to filter protocols that start with the letter P:

- 1) Choose Add AND from the left pop-up menu. Doing so will create two branches out of the predicate.
- 2) For the top branch, choose Kind from the left pop-up menu.
- 3) For the top branch, choose = from the right pop-up menu.
- 4) For the top branch, enter Protocol in the text field.
- 5) For the bottom branch, choose Name from the left pop-up menu.
- 6) Choose starts with from the right pop-up menu
- 7) Enter P in the text field.

## Adding and Removing Files to Track

When you create a class model file, you choose the file groups you want Xcode to track. The classes in the tracked groups are the classes that appear in the class model. When you add source code files to the tracked file groups, the classes in the new source code files automatically appear in the class model. If you forgot to add a file group when you created a class model file, don't worry. You can add file groups after creating the class model file.

To see the file groups Xcode is tracking in the class model, open the information panel and click the Tracking tab. You'll see the groups in your project that Xcode is tracking. To add a group, click the + button. A sheet with your project's groups will open. Select a group from the list and click the Add Tracking button. Selecting a group from the tracking list and clicking the minus button removes the group from the class model.

## Modeling Data

Xcode's data modeling tool is a lot more powerful than the class modeling tool. You can actually model data with the data modeling tool, not just view the data. Use the data modeling tool to create your program's data visually instead of writing code.

If you've taken a computer science course on database systems, you're familiar with entity-relationship modeling, which is what Xcode's data modeling tool uses. For those of you who haven't taken a database systems course, there are three terms you must know before you can model data: entities, attributes, and relationships. Entities are the basic building blocks of your data models. Entities consist of attributes and relationships. Attributes contain data. Relationships represent relationships to other objects. In programming terms, entities are your classes, attributes are your data members, and relationships are your methods.

## Adding a Data Model File to Your Project

If you create a Core Data application project, Xcode adds a data model file for you. If you have an existing project and want to use the data modeling tool, you must add a data model file to your project.

- 1) Choose File > New File.
- 2) Select Data Model from the file list and click the Next button.
- 3) Name the file, select the targets you want to add the class model to, and click the Next button.
- 4) Add the classes you want to appear in the data model file and click the Finish button. To add all your project's classes, select the project name and click the Add All button.

## Data Model Window

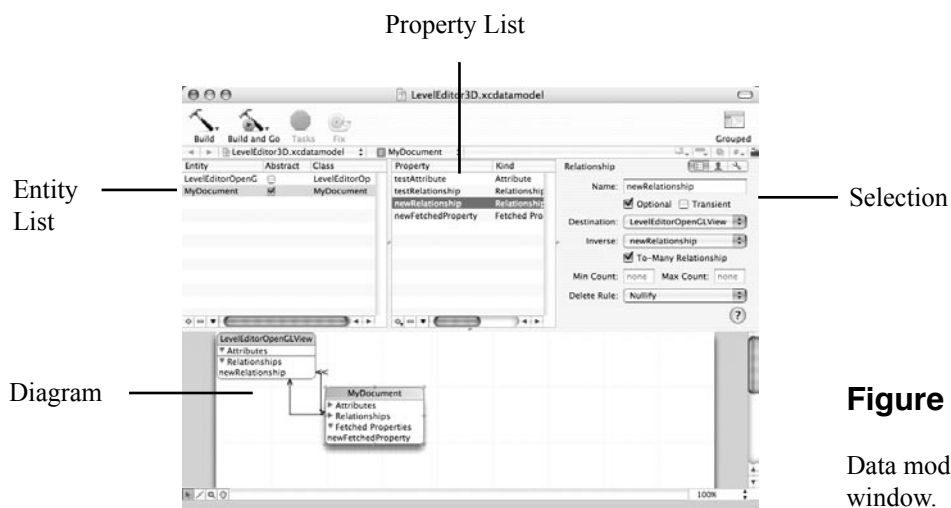
The data model window, shown in Figure 1.21, is where you examine your data models. The window has four sections.

- Entity list
- Property list
- Selection area
- Diagram

## Entity List

The entity list contains the entities in the data model. For each entity the list tells you the entity's name, the class it belongs to, and whether or not the entity is abstract. The class an entity belongs to must be `NSManagedObject` or a subclass of `NSManagedObject`.

Abstract entities are entities you don't create instances of in your program. They are meant to be inherited by other entities. Cocoa's `NSObject` class is an example of an abstract entity. You don't create objects in your Cocoa programs, you create the entities that inherit from `NSObject`: windows, views, controls, etc.



**Figure 1.21**

Data model window.



## Property List

Selecting an entity from the entity list fills the property list with the entity's properties. The property list tells you the property's name and the kind of property it is: attribute or relationship.

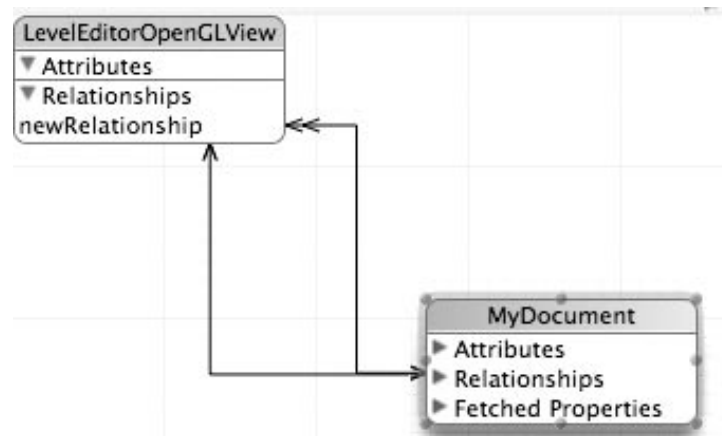
## Selection Area

Selecting a property from the property list fills the selection area with information about that property. For an attribute you can set its data type, minimum value, maximum value, and default value. For relationships you can specify the type of relationship and the relationship's destination.

## Diagram

The diagram area, which you can see in Figure 1.22, shows the entities and their relationships to each other. Lines represent the relationships between entities. The arrow points to the destination. Lines with arrowheads on both ends indicate a bidirectional relationship. A line with a single arrowhead indicates a to-one relationship, which means there is only one destination object. A line with a double arrowhead indicates a to-many relationship, which means there can be multiple destination objects.

Each entity in the diagram has at least three compartments: the entity name, attributes compartment, and relationships compartment. If an entity has fetched properties, there will be a fourth compartment for the fetched properties. Initially Xcode shows the class name, attributes compartment, and relationships compartment, but leaves the class's attributes and relationships hidden. Click the disclosure triangles to show the attributes and relationships in their respective compartments. Choosing Design > Roll Up Compartments shows the class name only. Choosing Design > Roll Down Compartments restores the attributes and relationships compartments.



**Figure 1.22**

Diagram area.

## Adding Entities

There are two ways to add an entity to the diagram. Choose Design > Data Model > Add Entity or click the + button in the entity list. After adding an entity, use the selection area to specify the following information about the entity:

- Its name.
- Its class. The class must be `NSObject` or a subclass of `NSObject`.
- The entity's parent.
- Whether or not the entity is abstract.

When you're starting out, giving each entity the class `NSObject` keeps things simple. Later on you may want to give each entity its own class. Each of these classes will be subclasses of `NSObject`.

## Adding Attributes

To add attributes to an entity, select the entity from the diagram or the entity list. Choose Design > Data Model > Add Attribute or click the + button in the property list and choose Add Attribute from the menu that opens when you click the button.

After adding an attribute, use the selection area to specify the attribute's name and data type. You can also specify whether the attribute is optional and whether it is transient. Core Data automatically stores your data in a data file and retrieves the data from the data file. Transient attributes are not stored in the data file. Transient attributes are the only attributes that can have an undefined data type.

When you specify the data type, what you can set depends on the data type you chose. For numerical and date attributes you can set the attribute's minimum, maximum, and default values. For Boolean attributes you can set the default value. For a string you can set the minimum length, maximum length, default value, and a regular expression. Use a regular expression to constrain the values the string can have. If you were using a string attribute to store a number, such as a credit card number, you would use a regular expression to limit the attribute to storing the characters 0–9.

## Adding Relationships

To add a relationship, select the line tool in the bottom left corner of the data model window. Use the line tool to draw a line from the source entity to the destination entity. After adding the relationship, use the selection area to specify the details of the relationship. You can set the following properties for a relationship:

- Name.
- Optional. An optional relationship does not require a destination.
- Transient. Transient relationships are not stored in the data file Core Data uses to store your data.
- Destination. You shouldn't have to change the destination. Choosing No Destination Entity from the pop-up menu removes the relationship from the diagram.
- Inverse, which bidirectional relationships use. If you have a relationship from entity A to entity B, the inverse is the corresponding relationship from B to A.
- To-Many. When you have a to-many relationship, the destination can have more than one object. An example of a to-many relationship is a student enrolling in courses. A student can enroll in more than one course.
- Min and Max Counts let you set the minimum and maximum number of destination objects in the relationship.
- Delete rule.

If you don't select the To-Many Relationship checkbox, the relationship is a one-to-one relationship. You cannot set the minimum and maximum counts for one-to-one relationships.

Delete rules determine what happens to the source and destination objects of a relationship when you delete the source object. There are four delete rules.

- No Action. Delete the source object but don't delete the destination objects.
- Nullify. Delete the source object. Do not delete destination objects, but set each destination object's inverse relationship to NULL.
- Cascade. Delete the source object and all destination objects.
- Deny. Do not allow the deletion if the source object has destination objects.

## Adding Fetched Properties and Fetch Requests

Fetch properties and fetch requests are two special properties an entity can have. A fetched property is a special type of relationship. Supply a set of conditions. The fetched property contains the related objects that meet the conditions you supply. A fetch request is an object that tells Core Data the data you want to find. Create a fetch request to retrieve data from an entity.

To add fetch properties and fetch requests to an entity, select the entity from the diagram or the entity list. Choose Design > Data Model > Add Fetch Property to add a fetch property. Choose Design > Data Model > Add Fetch Request to add a fetch request.

## Editing Predicates

Predicates are the way you specify conditions when searching. Fetched properties and fetch requests use predicates. Select a fetch property or fetch request from the browser and click the Edit Predicate button to open the predicate builder.

The predicate builder comes with one predicate for you. If you want to combine multiple conditions, click the + button to add a condition. There are two ways to create a predicate. The first way is to choose Expression from the left pop-up menu. The second pop-up menu disappears. Use the text field to enter the condition.

The second way is to choose Select Key from the left pop-up menu. The term key is misleading. You're going to be selecting an attribute. A dialog box opens for you to choose an attribute. The second pop-menu up contains conditions. The conditions depend on the attribute's data type. Numerical attributes have mathematical conditions like greater than, less than, and equal. String attributes have additional conditions such as starts with, ends with, and contains. Use the text field to specify the value.

Suppose you have an Employee entity with three attributes: first name, last name, and salary. You want to find the employees whose last names start with J and earn more than \$50,000 a year. To create this predicate:

- 1) Choose Add AND from the left pop-up menu. Doing so will create two branches out of the predicate.
- 2) For the top branch, choose Select Key from the left pop-up menu.
- 3) Select last name from the dialog box.
- 4) For the top branch, choose starts with from the second pop-up menu.
- 5) For the top branch, enter J in the text field.
- 6) For the bottom branch, choose Select Key from the left pop-up menu.
- 7) Select salary from the dialog box.
- 8) Choose greater than from the second pop-up menu.
- 9) Enter 50000 in the text field.

## Setting Information Dictionary Entries

All elements except fetch requests can have an information dictionary. This dictionary consists of key-value pairs. The information dictionary lets views and controllers access the model's properties.

To modify the information dictionary, click the User Info button in the selection area. The User Info button is the middle button in the three button group at the top of the selection area.. Click the + button to add a dictionary entry. Give the entry a key and a value.

## Adding Configurations

A configuration is a collection of entities. To edit configurations, select an entity from the diagram or the entity list. Click the configuration button in the selection area. The configuration button is the right button in the three button group at the top of the selection area. Click the + button to create a configuration. To add the selected entity to the configuration, select the checkbox next to the configuration name.

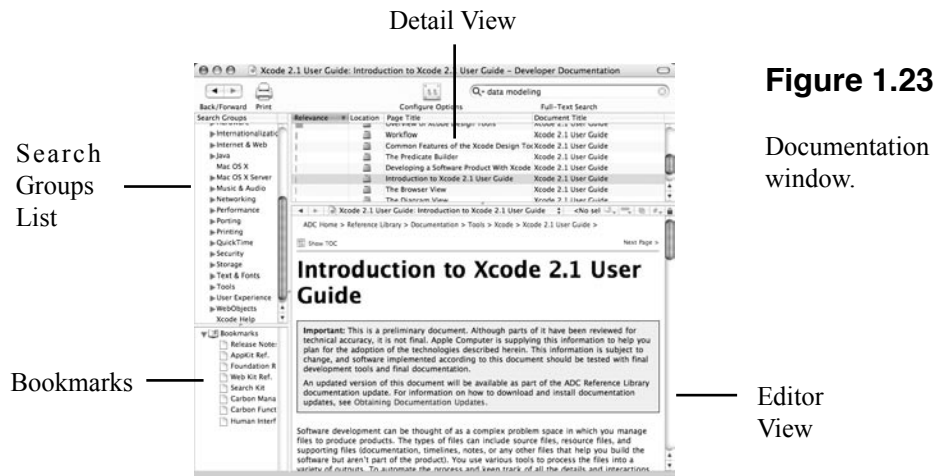
## Creating Source Code

If you create a subclass of `NSObject`, you usually write accessor functions for the class's properties. The data modeling tool can create method declarations and implementations for you. Choose Design > Data Model > Copy Method Declarations to Clipboard to create method declarations. Paste the method declarations in the header file. Choose Design > Data Model > Copy Method Implementations to Clipboard to create method implementations. Paste the implementations in the implementation file.

## Reading Developer Documentation

If you look at the `Developer` folder on your hard disk, you will see a folder called `ADC Reference Library` that contains developer documentation. You can read the developer documentation in Xcode, which comes in handy when you're writing code and you want to check something in the documentation.

To read the developer documentation, choose Help > Documentation to open the documentation window. The documentation window, shown in Figure 1.23, has four areas.



- The Search Groups list contains documentation groups to quickly reach the documentation you need.
- Bookmarks list.
- The detail view, which contains search results and symbols like function names and class names.
- The editor view, which is where you read the documentation.

## Searching the Developer Documentation

You can perform two kinds of documentation searches: API searches and full text searches. The pull-down button in the search field lets you choose the type of search. Use the API search to find information about a particular function. Use the full text search to read about a specific topic in the developer documentation.

If you spend more time reading the developer documentation than searching it, you'll want to use a full text search. When you select a documentation group from the Search Groups list, Xcode reads all the symbols in the group and places them in the detail view if Xcode is set to do an API search. Each documentation group has tens of thousands of symbols, and these symbols take time to load. No symbols load if you're performing a full text search. Perform API searches only when you want to actually search the API.

## Search Groups List

Mac OS X is a large operating system that requires lots of documentation to write software for it. The Search Groups list lets you filter the documentation so you can get to the documentation you're looking for. The Reference Library group contains all the documentation. If you select the Reference Library group, the editor view provides you with a map to the various sections of the developer documentation.

Clicking the disclosure triangle next to the Reference Library group reveals the documentation subgroups. You'll see a subgroup for each documentation section shown in the Reference Library area as well as a special group for Xcode help. The subgroups provide a way to quickly reach an area of the documentation. If you're writing a Cocoa program, all you care about is the Cocoa documentation so you can hit the Cocoa subgroup to reach the Cocoa documentation. If a subgroup has a disclosure triangle next to it, clicking it reveals more sections of documentation. You can reach any area of the documentation in two clicks.

## Bookmarks List

Below the Search Groups list is a list of bookmarks. Bookmarks let you quickly reach a specific piece of documentation. Choosing Find > Add to Bookmarks adds the current page to the bookmark list. Option-click a bookmark to change its name. Control-click a bookmark and press the Delete key to remove a bookmark from the list.

## Detail View

The detail view contains the results of your documentation searches. If you perform a full text search, the detail view contains an entry for each page of documentation that matches what you searched for. Each entry in a full text search contains four columns of information.

- Relevance.
- Location, which can be either your hard disk or Apple's developer site.
- Page Title.
- Document Title.

If you perform an API search, the detail view contains an entry for every symbol that matches the search. If you don't enter anything to search, all symbols in the selected documentation group match. When all symbols match, you get a lot of entries because each documentation group has lots of symbols. The Cocoa group has symbols for every class and function in the Cocoa framework. Each entry in an API search has four columns of information.

- The Symbol column provides the name of the symbol, which may be a function name, a class name, a data type, or a constant.
- The Type column tells you what type of symbol the particular symbol is. Table 1.1 lists the most common symbol types.
- The Language column tells you the programming language for this symbol. It can be C, C++, Java, or Objective C.
- The Class column tells you the class the symbol belongs to. This column will be blank for C language symbols because C is not an object-oriented language.

Selecting an entry in the detail view displays the documentation for the entry in the editor view. If an entry's documentation is on Apple's website, selecting the entry loads the entry in your Internet browser window and switches you from Xcode to your Internet browser.

## **Editor View**

The editor view is where you read the developer documentation. Above the editor view is a navigation bar that looks remarkably similar to the navigation bar of editor windows. Although the two navigation bars look similar, the documentation window navigation bar only uses three controls: the back button, the forward button, and the recently viewed file list. The buttons on the right side of the navigation bar are inactive.

## **Working with Targets**

Xcode projects consist of one or more targets. A target is a set of instructions to build a final product from the files in your project. To get as much as you can out of Xcode, you need to learn about targets and what you can do with them. In this section you'll learn about the following target-related topics:

- Adding targets to your project.
- The types of targets you can add to your project.
- The build phases that make up a target.
- Inspecting and modifying target settings.

### **Adding Targets**

When you create a project, Xcode supplies one target for the project. You can usually get away with using the target Xcode provides, but there are instances where you may need to create multiple targets. Suppose you're writing a library and you want to write a test application to make sure the code you wrote for the library is correct. In this case you would have two targets: one target to build the library and one target to build the test application.

To add a target to one of your projects, choose **Project > New Target**. The New Target Assistant window opens. It shows the types of targets you can add. I will discuss the types of targets you can add shortly. Select the target you want to make, and click the Next button. Give your target a name, and choose the project you want to add the target to. By default Xcode adds the target to the current project. Click the Finish button and you've created a new target.

## Special Targets

Most of the targets you can add correspond to Xcode's project types, but there are four special targets: aggregate, copy files, external, and shell script. Use an aggregate target to build a group of targets. The aggregate target depends on the targets that make up the aggregate. When you tell Xcode to build the aggregate target, it builds each target in the aggregate target.

The copy files target copies files to a specific location. Use a copy files target when multiple targets need to copy the same set of files.

An external target uses a program other than Xcode to build the target. External build projects have external targets.

A shell script target runs a shell script. Use a shell script target when multiple targets need to run the same shell script.

## BSD Targets

BSD targets use the BSD Unix APIs instead of the Carbon or Cocoa frameworks. You can create dynamic library, shell tool, and static library targets. A shell tool is a command-line program without a GUI.

## Carbon Targets

Carbon targets use the Carbon framework. You can create application, dynamic library, framework, loadable bundle, object file, resource file, shell tool, and static library targets.

## Cocoa Targets

Cocoa targets use the Cocoa framework. You can create application, dynamic library, framework, loadable bundle, object file, shell tool, and static library targets.

## Java Targets

Java targets use the Java frameworks instead of Apple's frameworks. You can create applet, application, package, and tool targets. A package is a collection of classes. You can think of a package as the Java equivalent of a framework.

## Kernel Extension Targets

There are two kernel extension targets. The Generic Kernel Extension target creates a kernel extension, and the IOKit Driver target creates a IOKit driver. You're more likely to create a kernel extension project than a kernel extension target.

## Legacy Targets

A legacy target uses the Project Builder build system that predates Xcode. Legacy targets cannot use Xcode features like ZeroLink and Fix and Continue. The main reason to use a legacy target is to maintain compatibility with old projects created with Project Builder, Xcode's predecessor.

## Unit Testing Targets

Xcode 2.1 introduced unit test bundle targets for Carbon and Cocoa programs. There are many unit testing tools available to automatically test your program's functions. Creating a unit test bundle target makes using unit testing tools easier in Xcode.

After creating a unit test bundle target, there are four tasks you must perform to unit test your program.

- Add the unit testing framework to your project.
- Add source code files and your project's product (application, library, or framework) to the target.
- Add your project's original target as a direct dependency for the unit test target.
- Write your test cases.

Most unit testing tools are frameworks that you add to your project. Choose Project > Add To Project to add the unit testing framework to your project.

Adding a unit testing framework isn't going to do anything until you add source code files to test. To add your project's source code files and product to the target:

- 1) Choose the unit test target from the Active Target pop-up menu.
- 2) Choose your project from the Groups and Files list.
- 3) Select the target checkbox for each source code file you want to test and your project's product.

To be able to unit test your program, Xcode must build your program before building the unit test target. In this scenario your program is a direct dependency of the unit test target. To add a direct dependency:

- 1) Select the unit test target from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the General tab in the information panel.
- 4) Click the + button to add a direct dependency.
- 5) A sheet with a list of your project's targets opens. Select a target from the list and click the Add Target button.

Now all you have to do is write the test cases that test your program's functions. Writing test cases is beyond the scope of this book.

## Duplicating Targets

Duplicate a target if you want to create a new target that differs slightly from an existing target. To create a copy of an existing target, select the target you want to duplicate from the Groups and Files list. Choose Edit > Duplicate to make the copy.



## Target Build Phases

Build phases are steps Xcode takes to build a target. Table 1.2 lists the possible build phases. Each target type has its own set of build phases. Click a target's disclosure triangle in the Groups and Files list to see the target's build phases.

### Adding Build Phases to a Target

Choose Project > New Build Phase to add a build phase to a target. The Copy Files, Run Scripts, and Build Resource Manager Resources build phases are the ones you're most likely to add. Use the Copy Files build phase when your program needs a framework or a library to be in a specific location. Use the Run Scripts build phase if your program includes files written in languages that Xcode doesn't directly support. Write a shell script to compile the files that Xcode can't compile for you.

Use the Build Resource Manager Resources build phase for Carbon programs that use resource files. If you're going to use resource files in your program, make sure each file has the extension `.r` or `.rsrc`. When you give each resource file the proper extension, Xcode merges them into one resource file that automatically loads when your program launches.

If you read the "Special Targets" section, you remember that Xcode has copy files and shell script targets. When should you use a target and when should use a build phase? Use a target when multiple targets need to copy the same set of files or run the same shell script. Use a build phase when one target needs to copy files or run a shell script.

**Table 1.2 Build Phases**

Build Phase	Description
Copy Files	Copies files from the project directory to a location you specify.
Run Scripts	Executes shell scripts when building the project. Some versions of Xcode call this phase Shell Script Files.
Copy Headers	Copies header files to the proper location in the final product. If you're creating a framework, you need the Copy Headers build phase. Some versions of Xcode call this phase Headers.
Copy Bundle Resources	Copies files that support your source code files, such as nib files, property list files, and image files, to the proper location in the final product. Some versions of Xcode call this phase Bundle Resources.
Compile Sources	Compiles source code files. Some versions of Xcode call this phase Sources.
Compile AppleScripts	Compiles AppleScript scripts. Some versions of Xcode call this phase AppleScript.
Link Binary with Libraries	Links the object files Xcode creates during compilation to the frameworks and libraries you added to your project. Some versions of Xcode call this phase Frameworks and Libraries.
Build Java Resources	Copies files that support your Java source code files to the proper location in the final product. Some versions of Xcode call this phase Java Resources.
Build Resource Manager Resources	Compiles resource files that contain Resource Manager resources. Some versions of Xcode call this phase Resource Manager Resources.

## Reordering Build Phases

When Xcode builds your project, it goes through the build phases in the order shown in the Groups and Files list. The default order works in most cases, but if you need to change the order, select a build phase and drag it to where you want it to appear in the build order. Be careful when ordering build phases. If you place build phases in the wrong order, Xcode won't be able to build your project correctly. Placing the Link Binary with Libraries build phase before the Compile Sources build phase will cause problems.

## Moving a File to a Different Build Phase

When you add a file to your project, Xcode automatically places it in a build phase. Normally Xcode places the file in the right build phase, but sometimes you need to move a file to a different build phase. If you add a Resource Manager resource file to your project, Xcode adds it to the Copy Bundle Resources build phase. You must move the file to the Build Resource Manager Resources build phase so Xcode can compile the file properly. Placing a resource file in the Build Resource Manager Resources build phase tells Xcode to compile the file and merge it with any other resource files into one resource file that automatically loads.

Clicking the disclosure triangle next to a build phase shows the files that are in that build phase. To move a file to a different build phase, select the file and drag it to the build phase where you want it to appear.

## Inspecting Target Settings

Xcode targets have lots of settings for you to configure. To inspect and edit a target's settings, open the target's information panel by selecting the target from the Groups and Files list and clicking the Info button. The information panel has five tabs.

- General
- Build
- Rules
- Properties
- Comments

Legacy targets, external targets, and Java projects do not have the Build, Rules, and Properties tabs. To change a legacy target, an external target, or a Java project's settings, double-click the target in the Groups and Files list. A window will open for you to change the target's settings. Figure 1.24 shows the areas you can set for a target in a Java Swing application.

## General Panel

The general panel is where you specify the target's name and any direct dependencies the target has. A *direct dependency* is a target Xcode must build before it builds the current target. You don't have to worry about direct dependencies unless your project has multiple targets. Aggregate targets require direct dependencies. The direct dependencies are the targets that make up the aggregate target.

Another common case where a target uses direct dependencies is when an application requires a library or framework to be built before building the application. The library or framework's target is the direct dependency for the application's target.

To add a direct dependency to a target, click the + button. A window opens that has a list of your project's targets. Select a target from the list and click the Add Target button to create the dependency.

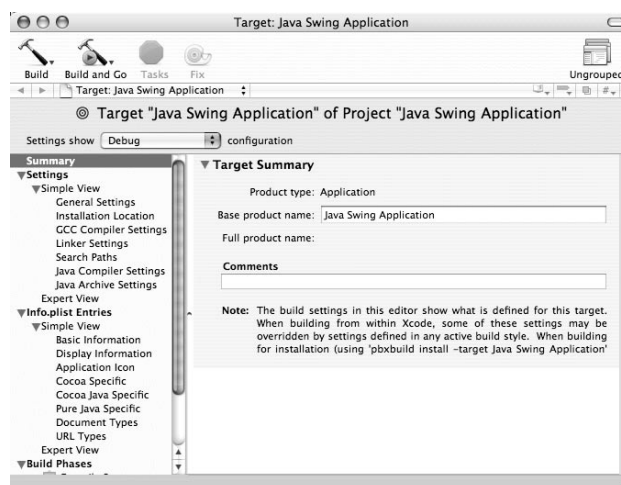
## Build Panel

The build panel is where you configure the build settings for a target. There are two places you can configure build settings: the target's information panel and the project's information panel. Both information panels have the same settings to configure. If you're running Xcode 2.1, configuring build settings from the target and project's information panels has the same effect. Configure the build settings where you wish.

For those of you running Xcode versions earlier than 2.1, configuring build settings can be confusing. Configuring a build setting for the active build style from the project's information panel overrides the same setting in the target's information panel. The active build style is the style Xcode uses to build your project. Target settings that have been overwritten by the active build style have a line through them. Refer to the "Build Settings" section later in the chapter to learn about the various build settings.

If you're running an old version of Xcode, where should you configure a build setting, in the target or in the build style? Configure a build setting in the target when you want the setting to be the same every time you build your program. An example of a build setting you want to configure in the target is the product name, which is the application's name for application projects. You want the product name to be the same every time you build your project, which is why you should set the product name in the target.

Configure a build setting in the build style when the setting is likely to change. An example of a build setting to configure in the build style is the code optimization level. When you start writing your program, you don't want any code optimization because optimized code is harder to debug. After you correct the errors in your program, you'll want to raise the code optimization level to make your code run faster.



**Figure 1.24**

Settings window for a non-native target.

## Rules Panel

The rules panel, shown in Figure 1.25, is where you specify the programs Xcode should use to compile files. A lot of the rules are no-brainers; Xcode has rules to compile `lex` source files with `lex`, `yacc` source files with `yacc`, and Rez resource files with Rez.

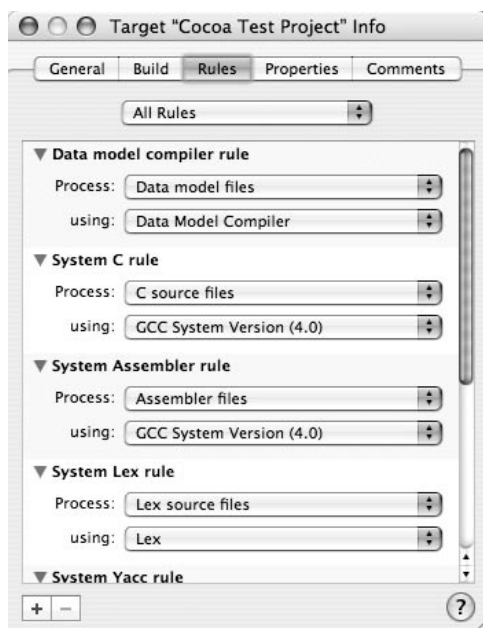
C, C++, and Objective C programs use rules to determine the `gcc` compiler version Xcode uses to compile your source code files. If you're using Xcode 2 on Mac OS X 10.4, `gcc 4` is the default compiler. To create universal binaries that run on both PowerPC and Intel hardware, use `gcc 4` as your compiler.

There is one downside of using `gcc 4`. C++ programs compiled with `gcc 4` won't run on versions of Mac OS X earlier than 10.3.9. Switching the compiler version to `gcc 3.3` allows your program to run on Mac OS X 10.2 and 10.3.

If you try to change one of the rules Xcode supplies for your project, an alert opens. The alert tells you that you must make a copy of the rule before changing it. Click the Make a Copy button to create the copy. Use the copy to change the rule.

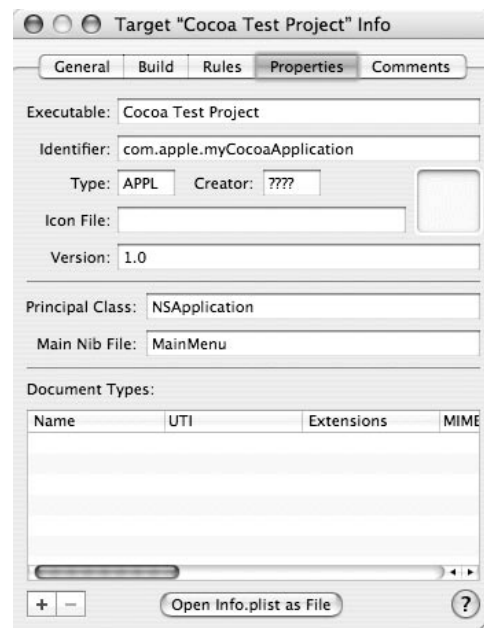
## Properties Panel

The properties panel, shown in Figure 1.26, is where you modify the target's property list. The property list tells the Finder information about the program such as its name, its icon, its version, and the types of document files it can open. The properties panel has three areas. At the top are general settings. In the middle are Cocoa-specific settings. At the bottom are the document files the program handles.



**Figure 1.25**

Target rules panel.



**Figure 1.26**

Target properties panel.

## General Property Settings

The Executable field is the name of the target. For an application you would enter the application name you want users to see in the Finder. The Identifier field is a string that uniquely identifies your program; Mac OS X uses it to create preference files. Apple recommends the identifier take the form:

```
com.CompanyName.ProgramName
```

The Type field is a four-character code identifying the kind of target: application, bundle, framework, library, etc. I can't see why you would need to change the target type. You'd be better off creating a new target.

If you designed an icon for your program, you would type its file name in the Icon File field. If you don't specify an icon file, you will get a default icon.

The Creator field is a four-character code identifying your application as the creator of any document files the application creates. Only worry about the Creator field if your application saves files. When the user opens a file in your application, you want the Open File dialog box to show only the types of files your application can open. Assume the application opens only the files it creates. Other operating systems use file extensions to identify the application that created the file. Mac users have the freedom to give files any name they want. The file name may have a different extension than the developer intended or no extension at all. The creator code tells the operating system your application made the file so you don't have to worry about the user changing the name of the file.

Xcode initially gives your program a creator code with four question marks. The four question mark code stands for all files, which means all the files a user has in a folder will appear in the dialog box when he or she tries to open a file. If you're developing an application for personal use, the creator code you give your application doesn't matter. If you release your application to the outside world, you should register the creator code at Apple's developer site to make sure it doesn't conflict with other applications. Your creator code must have at least one uppercase letter. After creating your creator code, submit the code to Apple. If another application registered the code, Apple will let you know, and you can submit another code. Otherwise, Apple will add your code to its database and send you an email confirmation.

The Version field lets you specify a version number for your program. Don't worry about program versions until you're close to finishing the program.

## Cocoa-Specific Settings

Only Cocoa programs use the Principal Class and Main Nib File fields. In most cases the principal class is `NSApplication`. The main nib file is the nib file the application loads when the user launches the application.

## Document Types

The Document Types table lists the file types your program supports. Table 1.3 lists the information you can set for document types. To add a document type, click the + button in the lower left corner of the panel. To remove a document type, select it and click the minus button. To modify any of the columns besides Role and Package, double-click an entry and type in the new information. Use the pop-up menu to change the role, and use the checkbox to change the package information.

Table 1.3 Document Type Fields

Column	Description
Name	The name you want to give to the document type. A Quake level editor might have the document name Quake Level.
UTI	A list of uniform type identifiers (UTI). A UTI is a string that uniquely identifies an abstract type, such as a file format.
Extensions	The file extension for the document type. Don't place a period before the extension. Adobe Acrobat would have a document type with extension pdf.
MIME Types	List of MIME (Multimedia Email Extensions) file types, file types a web browser uses. A PDF file has the MIME type <code>application/pdf</code> and a JPEG file has the MIME type <code>image/jpeg</code> .
OS Types	Four-character codes for the document's file type.
Class	The subclass of the Cocoa <code>NSDocument</code> class. Only Cocoa applications have to worry about the Class field.
Icon File	If you create an icon for document files, put the icon file's name here. Documents without an icon file will have the default Apple document icon.
Role	Documents can have three possible roles. Editors can view, edit, and save documents. Viewers can view documents, but can't edit or save them. Documents with the None role can't view, edit, or save.
Package	If the Package checkbox is selected, the document is a file package. Otherwise it's a single file.

## Configuring the Compiler

Programmers need to compile their programs in different ways. Some programmers want the compiler to generate the fastest possible code. Other programmers want the compiler to detect every possible problem in their code. You may even want to compile your program in different ways as you move further along in the development process. To accommodate the various ways people want to compile their code, Xcode has settings to configure the way it compiles programs.

### Build Configurations

A target is a set of instructions to build a final product from the files in your project. The way you want to build the final product changes as you get closer to finishing your program. When you start writing your program, you want to turn on debugging symbols and turn off code optimization so you can examine your program and correct any mistakes you made. When you have your code ready for release, you want to turn off debugging symbols to reduce the size of the final product and turn on code optimization to make your code run faster.

Build configurations solve the problem. They allow you to build the same target in different ways, reducing the need for you to create new targets. Targets tell Xcode *what* to build. Build configurations tell Xcode *how* to build the target. Xcode 2.1 introduced build configurations. Earlier versions of Xcode use build styles, which I cover next section.

Xcode supplies two build configurations for each project: debug and release. The debug build configuration contains settings that make debugging easier. The release build configuration contains settings to build a release version of your program. To look at your project's build configurations:

- 1) Select the project name from the Groups and Files list.
- 2) Click the Info button to open the information panel for the project.
- 3) Click the Build tab to look at the build configurations.

Figure 1.27 shows the information panel for build configurations. Use the Configuration pop-up menu to select the build configuration you want to view. Xcode initially sets your project to use the debug build configuration.

When you choose a build configuration from the Configuration pop-up menu, your choice is for viewing and making changes to that build configuration. Choosing a build configuration from the pop-up menu does not make the chosen build configuration the active one, the one Xcode uses to build your project. Choose Project > Set Active Build Configuration to change the active build configuration.

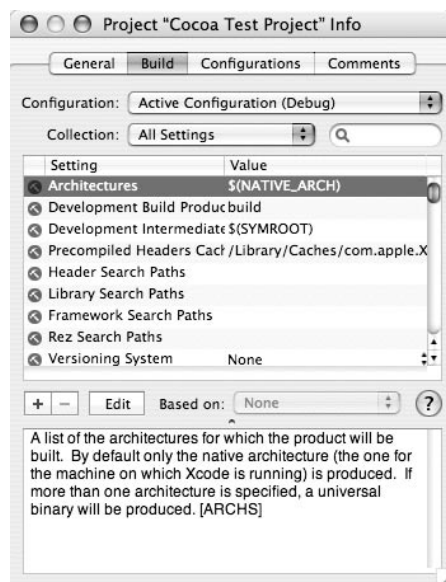
To add a build configuration to the list, click the Configurations tab. Select a configuration from the list and click the Duplicate button. Enter the name you want to give the build configuration. The debug and release build configurations Xcode supplies are sufficient for most of you, but here are some build styles you may want to add:

- A profiling build configuration, which generates profiling information for the `gprof` profiler.
- A code coverage build configuration, which generates information for the `gcov` code measurement tool.
- Build styles for different code optimization levels.

To delete a build configuration, select it from the list and click the Delete button.

## WARNING

Xcode gives you no warning dialog asking you if you're sure you want to delete a build configuration, even for the debug and release configurations Xcode makes for every project. Avoid accidentally clicking the Delete button.



**Figure 1.27**

Build configuration panel.

## Build Styles

If you're running a version of Xcode older than 2.1, you don't have build configurations. You have build styles. Build styles and build configurations do the same thing. They tell Xcode how to build a target.

There is one major difference between build styles and build configurations. Build configurations have one collection of settings you can modify from the target or from the project. Build styles have two collections of settings: one for the target and one for the build style. Changing a setting in a build style overrides the same setting in the target. Build styles are more confusing than build configurations.

To look at your project's build styles:

- 1) Select the project name from the Groups and Files list.
- 2) Click the Info button to open the information panel for the project.
- 3) Click the Styles tab to look at the build styles.

Use the Build Style pop-up menu to select the build style you want to edit as well as add, rename, and delete build styles. Xcode initially sets your project to use the development build style, the style that makes debugging easier. Choose Project > Set Active Build Style to change the active build style.

Choose New Build Style from the Build Style pop-up menu to add a build style to the list. A sheet will open, prompting you for a style name. The development and deployment styles Xcode supplies are sufficient for most of you, but here are some build styles you may want to add:

- A profiling build style, which generates profiling information for the `gprof` profiler.
- A code coverage build style, which generates information for the `gcov` code measurement tool.
- Build styles for different code optimization levels.

Choose Edit Build Styles from the Build Style pop-up menu to rename or delete a build style.

### WARNING

Xcode gives you no warning dialog asking you if you're sure you want to delete a build style, even for the development and deployment styles Xcode makes for every project. Avoid accidentally clicking the Delete button.

## Build Settings

Use build settings to control how Xcode builds your project. Xcode divides the build settings into five collections.

- General
- GNU C/C++ compiler
- Rez
- Lex scanner generator
- Yacc parser generator

Most Xcode projects have only the first two collections. I'm going to cover each build settings collection shortly. Use the Collection pop-up menu to choose a collection of build settings to view. Select a setting to read a description of the setting in the information panel.



Build settings have three types of controls. The type of control depends on the setting. Settings that can be either on or off have a checkbox. Select the checkbox to turn on the setting. Settings that have several possible values have a pop-up menu. Make your choice from the menu. The rest of the settings require you to enter text. To change the settings that require you to enter text, select the setting from the information panel and click the Edit button. A sheet opens for you to enter the setting's value. Click the OK button when you're finished typing.

If your version of Xcode uses build styles, remember that both targets and build styles have build settings. If a setting is blank for a build style, the setting may be set in the target. Settings you assign in the build style override the target's build settings.

If you're a beginning programmer, the number of settings can be intimidating, but don't freak out. Xcode configures the settings in a way that works well in general cases. You can stick with Xcode's default settings initially. If you find that Xcode isn't building your project the way you want it to, you can go back and configure the settings.

## **General Settings Collection**

The General Settings collection contains all the settings that do not involve compiling files. The collection has the following sections:

- Architectures
- Build Locations
- Search Paths
- Versioning
- Build Options
- Linking
- Packaging
- Deployment

### **Architectures**

The Architectures collection is where you tell Xcode the computer architectures it should build for. Initially Xcode builds products for the architecture on which you're running Xcode: PowerPC or Intel. To build a universal binary that runs on both architectures, select the Architectures setting and click the Edit button. A sheet opens. Select the Intel and PowerPC checkboxes and click the OK button.

If you need to create a 64-bit version of your program specifically for G5 Macs, double-click the Architectures setting's Value column. Enter the value `ppc64`.

Some versions of Xcode have the Architectures setting in the Build Options collection.

### **Build Locations**

When Xcode builds your project, it creates files. The files Xcode creates depends on the project type and the programming languages used. Some files that Xcode creates are object files, libraries, frameworks, and executable files. Use the Build Locations collection to specify where the files that Xcode creates appear. You can specify the directory where the final build product (application, library, or framework) appears as well as the directory where intermediate files like object files appear.

## Search Paths

The Search Paths collection is where you tell Xcode to search for header files, libraries, and frameworks. If you limit yourself to Apple's frameworks, you shouldn't need to change search paths. Programs that use third party libraries and frameworks must specify search paths.

## Versioning

The Versioning collection lets you determine whether or not your project should have a version number. If you choose to have a version number, use the Current Project Version setting to set the version number. Giving your project a version number is something you normally don't have to worry about until you get close to finishing your program.

## Build Options

Most of you will find the Generate Profiling Code setting to be the most interesting setting in the Build Options collection. Selecting the Generate Profiling Code checkbox tells Xcode to generate profiling code. If you're going to profile your code with `gprof` or `Saturn`, you must select the Generate Profiling Code checkbox.

If you're writing a framework or a library, you may be interested in the Build Variants setting. Use the Build Variants setting to build debugging and regular versions of your framework or library. Initially the build variant is `normal`, which means Xcode builds a normal application, library, or framework. If you want to add a debugging version of what you're building, select the Build Variants setting and click the Edit button. Enter `debug` in the sheet that opens and click the OK button.

## Linking

The Linking collection contains settings for the linker. When Xcode builds your project, the compiler compiles your source code files into object files. The linker links the object files with frameworks and libraries to create a final product, such as an executable file or a library.

The most interesting linking settings are ZeroLink and dead code stripping. When you enable ZeroLink, Xcode skips the linking stage and loads code when your program needs it. Dead code stripping removes code your program doesn't use from the executable file, giving you a smaller executable file.

Xcode 2.2 added several interesting linker build settings. Activating the Only Link In Essential Symbols build setting tells Xcode to copy only enough information for the debugger to do the work of the linker during debugging. This option speeds linking, but your program needs all of its object files to be available to debug the program.

The Display Mangled Names build setting tells Xcode to display mangled names for C++ symbols. Because multiple C++ functions can have the same name, the compiler mangles the function names to give each function a unique name. If you get link errors in your C++ program, activating the Display Mangled Names build setting can help you locate the source of the error.

Activating the Verbose Undefined Symbols setting tells Xcode to display additional information when an undefined symbol link error occurs. The additional information includes the file where the compiler found the symbol and tells you if the symbol is defined or referenced in the file.

## **Packaging**

The Packaging collection provides options on packaging the product Xcode creates when it builds your program. You can specify the executable name and any prefixes or suffixes you want to add to the name.

If you use Resource Manager resources in your program, selecting the Preserve HFS Data checkbox tells Xcode to preserve the resource forks when copying resource files. Mac OS X files have two forks: a data fork and a resource fork. But most Mac OS X files don't use the resource fork. If you place resources in a file's resource fork, you must select the Preserve HFS Data checkbox. If you don't select the checkbox, Xcode won't preserve the resource fork when it builds your program, and your program won't be able to find the resources stored in the resource file.

## **Deployment**

The Deployment collection determines how the final product is installed on the user's machine. You can tell Xcode where to install the product, the user that owns the product, the group that owns the product, and the file permissions to install the product files. For most of you, the Mac OS X Deployment Target setting is the most interesting setting in this collection. The deployment target is the oldest version of Mac OS X that can run your program.

## **Unit Testing**

Xcode 2.1 introduced the Unit Testing collection. Unit testing involves testing your program's functions to make sure they're correct. There are many tools available to make unit testing easier. The Unit Testing collection contains settings for using unit testing tools.

The Test Rig setting is where you specify the testing tool to use. Supply the path to the unit testing tool as the Test Rig setting's value.

Only application projects use the Test Host setting. The unit testing tool inserts the unit tests in the host. Supply the path to the application's executable file as the Test Host setting's value.

## **GNU C/C++ Compiler Settings Collection**

I'm sure you know that the GNU C/C++ Compiler Settings collection contains settings for the C and C++ compilers Xcode uses. What you may not know is that every Xcode project has a GNU C/C++ Compiler Settings collection, even Java and AppleScript projects. Even though all projects have settings for the C and C++ compilers, C, C++, and Objective C programs are the most likely to be modifying these settings. The GNU C/C++ Compiler Settings collection has the following sections:

- Language
- Code Generation
- Warnings
- Preprocessing

## Language

The Language collection contains miscellaneous settings that don't fit anywhere else. Some of the more interesting things you can do in the Language collection include the following:

- Choose the language compiler.
- Choose the language standard.
- Enable exception handling
- Set compiler flags.

The Compile Sources As setting is where you tell Xcode the language compiler to use: C, C++, Objective C, or Objective C++. Initially Xcode compiles your program's files according to the extension you give the files. Files with the `.c` extension use the C compiler. Files with the `.cpp` extension use the C++ compiler. Files with the `.m` extension use the Objective C compiler, and files with the `.mm` extension use the Objective C++ compiler. If you're writing in C you may want to compile your programs with the C++ compiler. The C++ compiler can compile C code, and it's stricter than the C compiler. Compiling your C code with the C++ compiler can catch errors you wouldn't find with the C compiler.

### NOTE

If you want to change the `gcc` compiler version Xcode uses, open the target's information panel and click the Rules tab. Click the + button to add a rule. Choose your language from the Process pop-up menu. Choose the `gcc` compiler version you want to use from the using pop-up menu.

The C Language Dialect setting is where you tell Xcode the language standard you want to use. The ANSI group defines the standards for the C and C++ languages. The `gcc` compiler adds extensions to these standards. Periodically (normally every 5-10 years) the ANSI group updates the language standards. Use the C Language Dialect setting to specify the language standard you want Xcode to use to compile your program.

The Enable C++ Exceptions setting turns on exception handling for C++ programs, and the Enable Objective C Exceptions setting turns on exception handling for Objective C programs. Exception handling is a way for C++ and Objective C programs to handle errors that occur when your program is running. If your program uses `try`, `catch`, and `throw` statements, make sure you turn on exception handling.

When you configure a compiler setting from the project information panel, you're indirectly setting command-line compiler flags. By using the information panel to configure compiler settings, Xcode saves you from having to enter dozens of compiler flags. Xcode provides a lot of compiler settings for you, but if you need to set a compiler flag that Xcode doesn't supply a setting for, use the Other C Flags and Other C++ Flags settings. C and Objective C programs should use the Other C Flags setting, and C++ and Objective C++ programs should use the Other C++ Flags setting. To set a compiler flag, select the appropriate setting and click the Edit button. Enter the flags in the sheet that opens and click the OK button.

## Code Generation

The Code Generation collection provides options on the machine code Xcode generates. Some things you can set in the Code Generation collection include:

- The code optimization level.
- Whether or not the code contains debugging symbols.
- The processor you want your code to run best on.

When Xcode compiles your program it translates the program from the language you used to write the program to machine language. Xcode performs a straight translation when there's no code optimization, making your code easier to debug. By using code optimization the compiler can take your code and make it run more efficiently.

As you're developing your program, making sure that your code contains debugging symbols is one of the most important things you can do. Debugging symbols are your program's variables and function names. If you want to use any of the profiling and debugging tools covered in this book, your program must generate debugging symbols. Without debugging symbols the tools display memory addresses instead of function and variable names.

If your code contains debugging symbols, use the Level of Debug Symbols setting to determine the level of debug symbol generation. If you want to use dead code stripping in your project, you must set the debug symbol level to All Symbols. Otherwise, the dead code stripping won't strip all the dead code.

The Instruction Scheduling setting tells Xcode to optimize instruction scheduling for a particular CPU. Initially Xcode optimizes instructions for your Mac's CPU.

Xcode 2.2 added one important build setting to the Code Generation collection. The Separate PCH Symbols build setting tells Xcode to create a separate file to store the debug symbols of a precompiled header. If you have this setting turned off, the debug symbols wind up in the binary file. The Separate PCH Symbols build setting gives you smaller binary files and faster link times.

To use the Separate PCH Symbols build setting, ZeroLink must be disabled and the Level of Debug Symbols build setting must be set to All Symbols.

## Warnings

The Warnings collection provides a series of checkboxes telling the compiler when to generate warnings. Compiler warnings tell you when you're doing something that's syntactically correct, but probably wrong. Turning on warnings is a good idea during development because the warnings let the compiler tell you about possible problems in your code that would be difficult to find without the warnings. Compiler warnings can save you hours during debugging.

Selecting the Treat Warnings as Errors checkbox tells Xcode to treat compiler warnings as compiler errors. If your code generates any compiler warnings, Xcode won't compile the program, forcing you to fix the warnings to build the final product. Treating warnings as errors forces you to write clean code from the start.

## Preprocessing

The Preprocessing collection is where you define and set the values for preprocessor macros. A macro performs text substitution. The following macro:

```
#define ARRAY_SIZE 500
```

Defines a macro `ARRAY_SIZE` that is a substitute for the value 500. Before compiling your program, the preprocessor goes through your code and replaces every instance of `ARRAY_SIZE` with the value 500.

The Preprocessing collection is for conditional macros. Suppose you add code to your program that writes debugging information to a file. As you're developing the program, you want to write the debugging information. When you have the code working properly, you want to stop writing the debugging information so your program will run faster. By defining a macro you can easily turn the debugging information on and off.

```
#define DEBUG

#ifdef DEBUG
    write debugging stuff
#endif
```

Instead of defining `DEBUG` in your code, you would define it in the Preprocessing collection. Defining `DEBUG` tells Xcode to write debugging information. Removing the `DEBUG` definition tells Xcode to skip over the debugging code. By defining `DEBUG` in the Preprocessing collection, you can turn the file writing on and off without having to recompile your code.

## Rez Settings Collection

Rez compiles Resource Manager resources. If your program uses Resource Manager resources, use the Rez Settings collection to modify the Rez compiler's settings.

## Lex Scanner Generator Settings Collection

`lex` (There's also a tool named `flex` that works similarly) is a tool to generate scanners. A scanner recognizes patterns in text and breaks those patterns into tokens, which you can think of as words. `lex` is a powerful tool for writing language compilers. If you were creating your own programming language, `lex` could take a source code file written in your language and break it up into its most basic pieces. Most of you won't have to worry about the `lex` settings.

## Yacc Parser Generator Settings Collection

`yacc` (There's also a tool named `bison` that works similarly) is a tool to generate parsers, programs that analyze the syntax of a file. `yacc` works with `lex`. It takes the tokens that `lex` generates and parses them, which is the computer language equivalent of grouping words into sentences and sentences into paragraphs. Most of you won't have to worry about the `yacc` settings.

## Adding Your Own Build Settings

After looking at all the build settings Xcode has, you might be wondering why you would need to add your own build settings. The information panel's list of build settings, while impressive, is not an exhaustive list. If you need to change a build setting that is not in the list, you must add the build setting.

The most common need to add build settings is to create universal binaries that run on old versions of Mac OS X. You must add separate build settings for the Intel and PowerPC deployment targets. Refer to the section "Building Universal Binaries for Older Versions of Mac OS X" later in this chapter for more information. Another reason to add a build setting occurs if your project has a Run Script build phase. You may want to define a build setting to use in your script.

To add a build setting:

- 1) Choose Customized Settings from the Collection pop-up menu.
- 2) Click the + button to create a new build setting.
- 3) Enter the setting's name in the Setting column.
- 4) Select the setting you created.
- 5) Click the Edit button.
- 6) Enter the setting's value in the sheet that Xcode opens and click the OK button.

The name of a build setting can have uppercase letters, underscore characters, and numbers in it. You cannot start a build setting name with a number. Choose Help > Show Build Settings Notes to see the entire list of Xcode's build settings.

To remove a build setting you created, select the setting and click the minus button.

## Setting Compiler Flags for One File

Most of the time you want to compile each file in your project with the same settings. Sometimes you may need to compile some files differently. By setting compiler flags specifically for one file, you can compile that file differently than the other files in the project. To set compiler flags for a single file:

- 1) Select the file in the project window.
- 2) Click the Info button to open the file's information panel.
- 3) Click the Build tab.
- 4) Enter the compiler flags in the Additional Compiler Flags text field.

## Java Compiler Settings

Xcode's build configurations do not contain build settings for the Java compiler. To view and modify the Java compiler settings, double-click the target in the Groups and Files list. The target's configuration window will open. Select Java Compiler Settings from the settings list on the left side of the window. The window will look like Figure 1.28. You can configure the following settings:

- The Java compiler to use.
- Whether or not to disable warnings.
- Whether or not to generate debugging symbols.
- The target version of the Java virtual machine.
- The source version of the Java Development Kit.
- The source code file encoding.
- Java compiler flags.

Use the Java Compiler pop-up menu to choose the compiler. Xcode gives you two Java compilers to choose from: `javac` and `jikes`. `javac` is Sun's Java compiler, and `jikes` is IBM's Java compiler.

The Warnings section has two checkboxes. Selecting the Disable warnings checkbox tells Xcode to disable any warnings your Java code generates. Compiler warnings tell you when you're doing something that's syntactically correct, but probably wrong. Selecting the Show usage of deprecated API checkbox tells Xcode to generate warnings when you use an outdated API that is not actively supported.

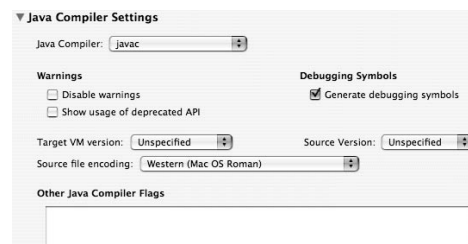
Selecting the Generate debugging symbols checkbox tells Xcode to generate debugging symbols. Debugging symbols are your program's variables and function names. If you want to debug your program or profile it with Shark, the program must generate debugging symbols.

Use the Target VM version pop-up menu to choose the version of the Java virtual machine you want to target. The Java virtual machine version determines the versions of Mac OS X that can run your program. You should target the earliest possible version of the Java virtual machine so more people will be able to run your program.

Use the Source Version pop-up menu to choose the version of the Java Development Kit (JDK) whose code the compiler accepts. You should use the highest possible source version. When you use a lower source version, the compiler won't accept Java features introduced in later versions of the JDK.

Use the Source file encoding pop-up menu to choose the character set your source code files use. Examples of character sets are Unicode, Western, Chinese, Japanese, Korean, Hebrew, Arabic, and Cyrillic. You normally don't need to change the source file encoding.

The Other Java Compiler Flags text field is where you enter any compiler flags that you can't set in the configuration window.



**Figure 1.28**

Java compiler settings.



## Configuration Files

Xcode 2.1 introduced configuration files. A configuration file is a text file that contains build settings. If you find yourself changing the same build settings for all your projects, you should use a configuration file. Add the settings you're always changing to the configuration file. Add the configuration file to your project and tell Xcode to base the build on the configuration file. Xcode will use the settings in the configuration file to build your project so you don't have to change the settings in the build configuration information panel.

### Creating a Configuration File

Because a configuration file is just a text file, you can create one in any text editor. Give the file the extension `.xcconfig` so Xcode knows the file is a configuration file. The easiest way to create a configuration file is to create it in Xcode. Choose **File > New File** to create a new file. Select **Configuration Settings File** from the list of file types. If your version of Xcode does not have the **Configuration Settings File**, select **Empty File in Project**, and make sure you give the file the extension `.xcconfig`.

When you create a new file, Xcode sets things up so the file is added to your project's targets. You do not want to add configuration files to targets. Deselect the checkbox next to each target in your project before clicking the **Finish** button to create the file.

Creating a configuration file in Xcode is easy, but you're not going to want to create a new configuration file for each project you create. The point of using a configuration file is to create a list of build settings once and use that list in multiple projects. Choose **Project > Add to Project** to add your configuration file to other projects. Make sure you do not add the configuration file to the other projects' targets.

Xcode projects can contain multiple configuration files. You can have one configuration file of debug build settings, a second configuration file of release build settings and use both files in your project. Use the debug configuration file in your debug build configuration, and use the release configuration file in your release build configuration.

### What Goes in a Configuration File?

A configuration file contains a list of build settings, with one setting per line. Each setting takes the following form:

```
SETTING = value;
```

Xcode has lots of build settings, too many for me to list here. Choosing **Help > Show Build Settings Notes** provides an overview of Xcode's build settings. Selecting a build setting in Xcode's build configuration information panel shows the formal build setting name in brackets. Build settings usually contain all uppercase letters. If you select the **Optimization Level** build setting, which is part of the **Code Generation** build settings collection, you will see the setting's formal name is `GCC_OPTIMIZATION_LEVEL`.

Suppose you want all of your projects to use the DWARF debugging format. You would add the following setting to the configuration file:

```
DEBUG_INFORMATION_FORMAT = dwarf;
```

Xcode 2.2 introduced the ability to add comments to a configuration file. Xcode uses the C++ `//` comment style.

```
// This is a comment.
```

A configuration file can contain as many build settings as you want, but don't put every build setting in the configuration file. Put only the build settings you don't want to be constantly changing. If there are only three build settings you're constantly changing, put those three settings in the configuration file. Xcode will use the settings in the build configuration information panel for the rest of the build settings.

## Telling Your Project to Use a Configuration File

After writing your build configuration files, you must add them to your project and tell Xcode what configuration file to use. If you haven't added configuration files to your project, choose Project > Add to Project to add them, making sure you don't add them to your project's targets.

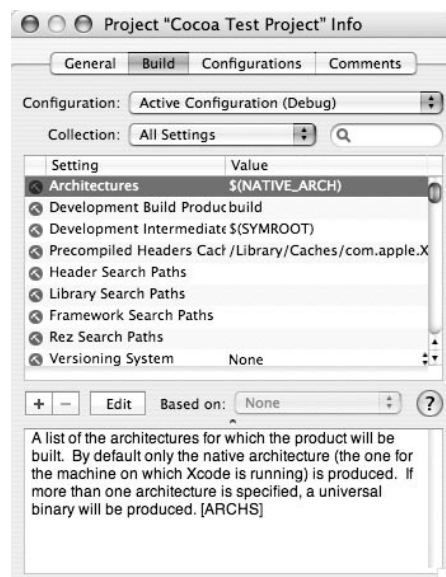
To assign a configuration setting file, open the build configuration information panel for your project. Refer to Figure 1.29 to see an example of the information panel. In the lower third of the panel is the Based on pop-up menu. The menu contains the configuration files you added to your project. Choose a configuration file from the menu. Make sure you choose a configuration file for each build configuration in your project, assuming you want to use a configuration file for all your project's build configurations.

## Overriding the Configuration File

Suppose you have a configuration file with 15 settings. You create a new project, but want to use only 13 out of the 15 settings in the configuration file. How do you tell Xcode to use only the 13 settings you want to use?

The solution is to use the build configuration information panel. When you change a build setting from the build configuration information panel, it overrides the setting in the configuration file. In the example from the last paragraph, you would open your project's build configuration information panel and change the two settings you want to override.

Because the build configuration information panel overrides the configuration file, be careful when you open the build configuration information panel. You don't want to accidentally override any of your configuration file's build settings.



**Figure 1.29**

Build configuration panel.

## Compiling Your Program

After writing the source code for your program, you'll want to compile the program to make sure everything works. Xcode calls the process *building*. When building your project, Xcode compiles your source code files, links them with the frameworks in your project, and creates the final product, such as an application or a library.

### Precompiled Headers

Xcode supplies a prefix file for Cocoa and Carbon application projects. The prefix file contains a list of header files. Xcode precompiles the header files in the prefix file. By precompiling these header files, Xcode has less work to do when it compiles the files that include the precompiled header files. If you have a lot of source code files that include a header file, adding the header file to the prefix file speeds up the building of your project.

What header files should you put in a prefix file? Header files that multiple source files include and header files that don't change often are the best files to add to a prefix file. The header files from Apple's frameworks are great files to put in a prefix file. If you're writing a Cocoa program, most of your files are going to include the Cocoa header file `Cocoa.h`, and you're not going to make changes to `Cocoa.h`.

Why do you want to place header files that rarely change in a prefix file? Xcode recompiles the precompiled header file when the prefix file changes or the header files in the prefix file change. If you add a header file that you're constantly changing, Xcode has to recompile the precompiled header every time you change the header file. Recompiling the precompiled header is slower than not using prefix files at all.

To add header files to a prefix file, open the prefix file. The prefix file of a Cocoa application has the name `ProjectName_Prefix.h`, and the prefix file of a Carbon application has the name `ProjectName_Prefix.pch`. The prefix files for Cocoa and Carbon applications include the Cocoa and Carbon header files, respectively. Use the `#include` statement to add other header files.

After adding files to your prefix file, make sure Xcode is set to precompile the headers in the prefix file.

- 1) Select the target name from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the Build tab in the information panel.
- 4) Choose Gnu C/C++ Compiler from the Collection pop-up menu.
- 5) Make sure the Precompile Prefix Header checkbox is selected.
- 6) Make sure the Prefix Header setting shows the name of your prefix file.

### ZeroLink

When you build your project with ZeroLink, Xcode links frameworks, libraries, and object code when your program needs them instead of embedding them into the final product. ZeroLink makes building faster because you're skipping the linking stage. Build speed is important in the early stages of development because you're going to be compiling often as you fix bugs and make other improvements to your code.

ZeroLink has one downside. The executable file Xcode builds is just a shell, which isn't a problem as long as you don't move the executable file. If you send a copy of the executable file to other people to run on their computers, the program won't run because the code the program needs is on your computer, not in the executable file. ZeroLink can also cause problems if you run your program on performance tools like Shark and MallocDebug.

Xcode's debug build style enables ZeroLink while the release build style disables ZeroLink. When your program is ready for release, make sure to build the project with the release build style. Building your project with ZeroLink disabled allows people to run your program on their computers.

Xcode 2.2 lets you turn off ZeroLink globally so you don't have to turn it off for every project you create. Choose Build > Allow ZeroLink to turn it off, and choose Build > Allow ZeroLink again to allow it in your projects. If you allow ZeroLink, the ZeroLink build setting determines whether or not Xcode builds your project with ZeroLink.

## Distributed Builds

One of the most common complaints about Xcode is its speed compiling programs. For those of you programming in C, C++, or Objective C on a network with multiple Macs, distributed builds can reduce the time Xcode takes to compile programs. Distributed builds use other computers on the network to compile source code files. The more computers you have on the network, the faster the compilation goes.

To distribute builds to another computer, that computer must be running the same operating system version and compiler version as your Mac. If you're building your program with gcc 4 on Mac OS X 10.4 and the other computer is running Mac OS X 10.3 and Xcode 1.5, you won't be able to distribute your build to the other computer. To distribute builds to other computers:

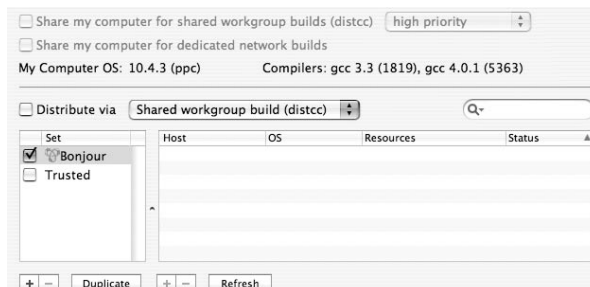
- 1) Choose Xcode > Preferences to open Xcode's preferences panel, which you can see in Figure 1.30.
- 2) Select the Distribute builds to checkbox.

When you select the Distribute builds to checkbox, the Bonjour build set checkbox gets selected too. By selecting the Bonjour checkbox, Xcode automatically lists the computers you're connected to. For each connected computer, Xcode tells you the computer's name, the operating system it's running, the compilers installed on that computer, and its status. You can distribute builds only to computers with Sharing status. Click the Apply button to have your builds distributed to every computer with Sharing status.

Having your builds distributed to every available computer on the network may not be what you want. You can create custom build sets that contain the computers you want to distribute builds to. Click the Duplicate button to duplicate an existing build set or click the + button to add a build set. Use the + and minus buttons under the host list to customize your build set. Select your build set from the build set list to distribute builds to the computers you want.

Xcode 2.3 introduced dedicated network builds. Dedicated network builds are designed to build projects using 12 or more Macs so most of you won't be able to take advantage of them. If you have enough Macs to use dedicated networked builds, choose Dedicated network build from the menu next to the Distribute via checkbox.

You can also choose to share your computer so other programmers can distribute their builds to your Mac. Select the Share my computer for building with checkbox to share your computer. Xcode 2.2 forces you to click the lock at the bottom of the distributed builds preferences panel to share your computer.



**Figure 1.30**

Distributed builds preferences.

## Cleaning Targets

Cleaning a target removes everything you previously built for a target. When you build a project with a clean target, you must recompile all your source code files so you don't want to clean a target unless it's necessary. Changing compiler settings in your project requires a clean target for the changes to take effect. Suppose you're compiling your files with the Fast optimization level and you change the optimization level to Faster to measure the speed difference between the two levels. To increase the optimization level you must recompile all the files, which requires you to clean the target.

Choosing **Build > Clean** cleans the active target. To clean all the targets in your project, choose **Build > Clean All Targets**.

## Building Your Project

Xcode provides four methods to build your programs.

- Compile a single file.
- Build a project.
- Build a project and run the program.
- Build a project and debug the program.

To compile a single file, select the file from the project window. Choose **Build > Compile**.

Choosing **Build > Build** builds the project. It compiles any source code files that changed since the last time you built the project. If you haven't built your project before or you cleaned the target, Xcode compiles all the source code files. After compiling the files, Xcode performs any additional steps required to create the target.

Choose **Build > Build and Run** to build and run your program. Building and running builds your project and runs it from Xcode. To be able to run your program, it must build successfully with no errors.

Choose **Build > Build and Debug** to build and debug your program. Building and debugging builds your project and runs it from Xcode's debugger. I cover debugging in the next chapter.

## Seeing More Build Details

When you build your program, the only thing Xcode tells you is whether the build succeeded or failed. If the build failed, you want to know what caused the build to fail. To see more details of a build, open the build results window, shown in Figure 1.31, by choosing **Build > Build Results**. The build results window has three areas.

- Build results.
- Build transcript. Initially the build transcript isn't visible. You must click the build transcript button, located below the build results, to see the build transcript.
- Editor, which lets you fix any mistakes in your code.

The build results area gives you a play-by-play account of the build, listing each step taken during the build along with any errors or warnings. Each build step executes commands from the command line. Xcode shields you from the details of executing commands, but the build transcript shows the executed commands. When you compile a file, the build results area tells you that Xcode compiled the file. The build transcript shows the commands Xcode called to compile the file. Selecting a build step from the build results area takes you to the selected step in the build transcript.

If you have an error during your build, select the error from the build results area. Xcode opens the file containing the error in the editor and takes you to the line of code with the error. From there you can locate the cause of the error and correct it.

If you want to have the build results window open automatically, click the build panel behavior button, which is next to the build transcript button. By clicking the button you can tell Xcode to open the build results window every time you build your project or when there are errors during the build. Choose Open Global Build Preferences to determine when Xcode opens the build results window for all your projects.

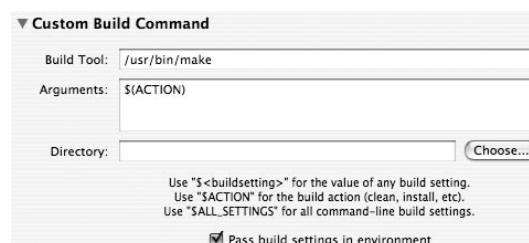
## Building for Unsupported Languages

Xcode natively supports projects written in AppleScript, C, C++, Java, and Objective C. It can also build projects written in languages Xcode doesn't natively support. The Xcode Tools include Perl, PHP, Python and Ruby. Other popular languages are Fortran, Lua, Pascal, and Smalltalk, but you have to download a compiler to use those languages. You can even create your own programming language and build programs written in that language from Xcode. All you need is a program to do the building.

To use a language that Xcode doesn't natively support, create an external build project. You must tell Xcode to use your language's build tool to build the project.

- 1) Double-click the target's name in the Groups and Files list. Doing so opens the target's settings window.
- 2) Select Build Tool Configuration to set the build tool, which you can see in Figure 1.32.
- 3) In the Build Tool text field, type the path to the program you're going to use to build the program. Initially for an external build project, the build tool is `make`, located at `/usr/bin/make`.
- 4) In the Arguments text field, enter any arguments you want to use to build the program. Normally this involves compiler flags you want to use when building the program.

When you build your project, Xcode uses the tool you specified to do the building. You won't be able to run or debug the program from Xcode, but you can build it without having to go to the command line.



**Figure 1.32**

External build configuration.

## Tips for Correcting Build Errors

One of the most frustrating aspects of writing Mac software for people new to Mac development is getting the program to compile. Compilers can be picky, spitting out error messages on code that worked on another compiler. In this section I provide some tips to fixing the errors you get when trying to build your program.

### Add All Necessary Frameworks

Missing frameworks are the leading cause of linker errors and cause compiler errors if your program calls functions from the missing frameworks. Sometimes you may need to include a framework you weren't even aware you needed. The GLUT framework takes care of operating system dependent portions of an OpenGL program like event handling and creating windows. The Mac OS X version of the GLUT framework is written in Cocoa. If you're writing a program that uses the GLUT framework, you must add the Cocoa framework to your project as well as the GLUT framework.

### Include Necessary Header Files

Adding a framework isn't the only step you must take if you want to call the framework's functions in your program. You must remember to include the framework's header files as well.

Mac OS X has a slightly different way of including system headers than other operating systems. The method Mac OS X uses to include system headers causes build errors if you're unaware of the method. Suppose you want to play QuickTime movies in your program. You add the QuickTime framework to your project and include the header file `Movies.h`.

```
#include <Movies.h>
```

This statement generates an error during building. Mac OS X requires you to enter the framework the header file belongs to before the header file when including a system header. The following line demonstrates the proper way to include the `Movies.h` header file:

```
#include <QuickTime/Movies.h>
```

### The Error May Not Be Where Xcode Says It Is

When Xcode finds a syntax error in your code, it reports the file where the error occurs along with the line number. You go to the line of code and can't find anything wrong. In this situation the error may have occurred in an earlier line of code. Check the lines of code above the line where Xcode reported the error.

### One Error Can Cause Multiple Syntax Errors

Sometimes one error can trigger multiple syntax errors. It can be overwhelming to look at the build transcript and see hundreds of error messages. Fix one error at a time. You may discover you have fewer mistakes in your program than you thought.

## Look for Typographical Errors

Typographical errors cause a high percentage of syntax errors. Make sure you're not missing semicolons. Make sure each left brace has a matching right brace. Make sure each left parenthesis has a matching right one. Check for misspelled variable and function names.

## Check Function Arguments

Improper function arguments cause many syntax errors. Check the arguments in your function calls. Make sure the number of arguments match, the arguments are in the right order, and the arguments have the proper data type. Pointer variables as arguments cause a lot of syntax errors. If you pass a non-pointer variable improperly, you get a syntax error.

Let's use an example to demonstrate passing a non-pointer variable improperly. The Core Foundation function `CFURLGetFSRef()` creates a file system reference from a file location. You would use the file system reference to open the file. The second parameter to `CFURLGetFSRef()` is a pointer to a file system reference. If you have the following code:

```
CFURLRef theFile;  
FSRef fileRef;  
Boolean success;  
  
// Code to retrieve the file location has been omitted.  
  
success = CFURLGetFSRef(theFile, fileRef);
```

You get an error because the function `CFURLGetFSRef()` takes a pointer to `FSRef` as an argument, and you passed a `FSRef`. You must pass the address of the variable `fileRef`.

```
success = CFURLGetFSRef(theFile, &fileRef);
```

## Running your Program

Choose **Debug > Run Executable** to run your program. The run log opens and your program launches. If you're writing a command-line program, the program's output appears in the run log.

Click the **Terminate** button in the run log toolbar to stop running your program. After terminating your program, the button changes to a **Run** button. Click the **Run** button to rerun your program. The run log has pop-up menus to pick the active target and active executable, the executable file Xcode launches when you run your program from the run log.



## Developing for Different Versions of Mac OS X

Xcode projects are set to create applications for the version of Mac OS X you're running. By choosing an appropriate deployment target and compiler version, you can create an application that runs on multiple versions of Mac OS X. But what do you do if you need to develop a version of your application specifically for a different version of Mac OS X than the one you're running? The Xcode Tools ship with software development kits (SDK) that contain the header files and libraries for previous versions of Mac OS X. By building your program with the Mac OS X 10.2 SDK, your program will run on Mac OS X 10.2 and later versions of OS X.

To be able to develop for different versions of Mac OS X (Xcode calls it cross-development), you must install the SDKs, which are on the Xcode Tools disc. If you performed an easy install of the Xcode Tools, you did not install the SDKs. You must perform a custom install to install the SDKs.

After installing the SDKs, you must perform three tasks in Xcode to develop for a different version of Mac OS X.

- Choose a SDK to develop for.
- Choose a deployment target.
- Supply a prefix file for each target.

### Choosing a SDK

Initially Xcode builds your program using the SDK of the Mac OS X version you're running. You must change the SDK to the one you want to use. To choose a different SDK:

- 1) Select the project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the General Tab in the information panel.
- 4) Choose a SDK from the Cross-Develop Using Target SDK pop-up menu.

### Choosing a Deployment Target

The deployment target is the earliest version of Mac OS X that can run your program. If you're using a specific SDK, the deployment target should match the SDK version. To choose a deployment target of Mac OS X for one of your project's targets:

- 1) Select the target from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the Build tab in the information panel.
- 4) Choose Deployment from the Collection pop-up menu.
- 5) Choose an operating system version for the Mac OS X Deployment Target setting.

## Supplying a Prefix File

For cross-development to work, you must supply a prefix file for each target that lets Xcode know the SDK you're using. If you use the prefix file that Xcode supplies when it creates your project, you should have no problems with cross-development. Some developers like to use the umbrella headers for the Carbon and Cocoa frameworks as their prefix files. They supply the path to the umbrella header as the prefix file. Using umbrella headers as prefix files doesn't work for cross-development because Xcode doesn't allow absolute paths for cross-development. Add the umbrella header to the prefix file.

```
#include <Cocoa/Cocoa.h>
```

After adding the umbrella header to your prefix file, tell Xcode to use the prefix file as your prefix header instead of the umbrella header.

- 1) Select the target name from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the Build tab in the information panel.
- 4) Choose Gnu C/C++ Compiler from the Collection pop-up menu.
- 5) Make sure the Prefix Header setting shows the name of your prefix file.

## Creating Universal Binaries for PowerPC and Intel Hardware

At the 2005 Worldwide Developers Conference Apple announced it was shifting from PowerPC to Intel processors for Macintosh computers. To smooth the transition Xcode 2.1 lets you create universal binaries that run on Macs using PowerPC and Intel processors. Creating a universal binary requires the following steps:

- 1) Install the Mac OS X 10.4 Universal SDK.
- 2) Use `gcc 4` as the compiler.
- 3) Use the Mac OS X 10.4 Universal SDK in your project.
- 4) Set a deployment target of Mac OS X 10.4 in your project.
- 5) When your program is ready for release, tell Xcode to build for both the PowerPC and Intel architectures.

The reason you need to compile with `gcc 4` and set a deployment target of Mac OS X 10.4 is to support Intel Macs. If you need to support earlier versions of Mac OS X on PowerPC Macs, you can create separate deployment targets for Intel and PowerPC and use different compilers to build the Intel and PowerPC versions of your program. Refer to the section "Building Universal Binaries for Older Versions of Mac OS X" for more information.

## Installing the Mac OS X 10.4 Universal SDK

The Mac OS X 10.4 Universal SDK contains the libraries you need to create universal binaries. If you're running Xcode 2.2, you should have the Universal SDK installed.

If you're running Xcode 2.1, you must install the Mac OS X 10.4 Universal SDK from the Xcode 2.1 disk. You must perform a custom install to install the SDK. The easy install does not install the cross-development SDKs.

## Selecting gcc 4 as the Compiler

Universal binaries for Intel Macs must be built with the gcc 4 compiler. To make sure you're using the gcc 4 compiler:

- 1) Select the target from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the Rules tab in the information panel.
- 4) Make sure the System C Rule is set to process C files with gcc 4.

If you're writing a C++ program, you should add a rule that tells Xcode to use gcc 4 to compile C++ files.

## Using the Mac OS X 10.4 Universal SDK in Your Project

After installing the Mac OS X 10.4 Universal SDK, you must tell Xcode to use it for your project. To tell Xcode to use the Mac OS X 10.4 Universal SDK:

- 1) Select the project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the General tab in the information panel.
- 4) Choose Mac OS X 10.4 Universal from the Cross-Develop Using Target SDK pop-up menu.

## Setting the Deployment Target to Mac OS X 10.4

Universal binaries for Intel Macs require you to develop your program for Mac OS X 10.4. To choose Mac OS X 10.4 as your deployment target:

- 1) Select the target from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Click the Build tab in the information panel.
- 4) Choose Deployment from the Collection pop-up menu.
- 5) There is a pop-up menu for the setting Mac OS X Deployment Target. Choose Mac OS X 10.4 from the menu.

## Building for PowerPC and Intel Architectures

As you're developing your application you don't need to create a universal binary. Just develop for the architecture you're using, which for most of you is PowerPC. Debug binaries run only on the architecture you're using. Building a debug version of a universal binary will fail.

When your application is ready for release, switch to the release build configuration and create the universal binary. To create the universal binary:

- 1) Select the target from the Groups and Files list.
- 2) Click the Info button to open the target's information panel.
- 3) Choose Release from the Configuration pop-up menu.
- 4) Click the Build tab in the information panel.
- 5) Choose Architectures from the Collection pop-up menu.

- 6) Click the Edit button in the information panel.
- 7) A sheet opens. Select the PowerPC and Intel checkboxes and click the OK button.
- 8) Choose Project > Set Active Build Configuration > Release.
- 9) Build your project.

Following the steps I detailed creates 32-bit binaries that can run on any Mac running a suitable version of OS X. If you add the `ppc64` architecture, Xcode will create a 64-bit binary for G5 Macs. Xcode 2.4 added the ability to create 64-bit Intel binaries. Add the `x86_64` architecture to create a 64-bit binary.

Most of Apple's frameworks, including the Cocoa and Carbon frameworks, do not have 64-bit support so there is no reason for most of you to create 64-bit binaries. If you try to compile a 64-bit version of a Cocoa application on Mac OS X 10.4, you will get a bunch of compiler errors. Unix scientific applications that need to do a lot of number crunching on Mac Pros or G5 Macs are the applications that would benefit most from 64-bit binaries on Mac OS X 10.4. 64-bit support is coming in Mac OS X 10.5 so there are benefits to knowing how to build 64-bit binaries.

## Building Universal Binaries for Older Versions of Mac OS X

Mac OS X 10.4 is the earliest version of Mac OS X that will run on Intel Macs, but you would like your universal binary to run on PowerPC Macs running older versions of OS X. To support earlier versions of Mac OS X on PowerPC, you must add the build setting `MACOSX_DEPLOYMENT_TARGET_ppc`. This setting specifies the earliest version of Mac OS X that can run the PowerPC version of your universal binary. The Intel version's deployment target will still be Mac OS X 10.4.

- 1) Open the build setting information panel.
- 2) Choose Customized Settings from the Collection pop-up menu.
- 3) Click the + button to create a new build setting.
- 4) Give the setting the name `MACOSX_DEPLOYMENT_TARGET_ppc`.
- 5) Give the setting the value of the operating system version you want as your deployment target. To run on Mac OS X 10.2 and later, use the value 10.2.

C++ programs require a little more work. Programs for Intel Macs must be built with `gcc 4`, but C++ programs built with `gcc 4` run only on Mac OS X 10.3.9 and later. To support earlier versions of Mac OS X, you must build the PowerPC version of your program with `gcc 3.3`.

- 1) Open the build setting information panel.
- 2) Choose Customized Settings from the Collection pop-up menu.
- 3) Click the + button to create a new build setting.
- 4) Give the setting the name `GCC_VERSION_ppc`.
- 5) Give the setting the value 3.3.

# Chapter 2

## Debugging with Xcode

After you eliminate the syntax errors in your code, you're going to spend a lot of time using Xcode's debugger. With the debugger you can go through your code line by line to make sure your program runs the way you expect. Use the debugger to find mistakes in your program that the compiler didn't find.

### Before You Debug

If you create a new project, write the code for the project, and build the project, you should be able to debug your program with no problems. But there are steps you can take before you debug to make debugging go more smoothly.

- Configure Xcode.
- Choose a debugging format.
- Set your program's debugging options.
- Set environment variables for debugging.
- Use the debug versions of frameworks.
- Use the Guard Malloc library.
- Configure remote debugging, where you run your program on one computer and debug it on a second computer.

### Configuring Xcode for Debugging

To debug your program you must tell Xcode to generate debugging symbols when building your program. The debugging symbols contain information the debugger needs, like your program's function names and variables. If you don't generate debugging symbols, you won't be able to debug your program.

Xcode's debug build configuration tells Xcode to generate debugging symbols. As long as you use the debug build configuration, you should have no problems. If you want to make sure you're generating debugging symbols, perform the following steps:

- 1) Select your project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the Build tab in the information panel. Some versions of Xcode have a Styles tab.
- 4) Choose Code Generation from the Collection pop-up menu.
- 5) Make sure the Generate Debug Symbols checkbox is selected.

While you're in the Code Generation collection, you should set the optimization level to None. Setting the optimization level to None ensures that every line of your code executes in the order you wrote it. During optimization the compiler may add, remove, or rearrange lines of code to make the program run faster, which makes debugging more difficult.

If you want to use Xcode's Fix and Continue feature, make sure the Fix and Continue checkbox is selected. The Fix and Continue checkbox also appears in the Code Generation collection. Fix and Continue lets you make changes to your source code without leaving the debugger. If you're writing a C++ program, you must also enable ZeroLink to use Fix and Continue. The ZeroLink checkbox appears in the Linking collection.

## Choosing a Debugging Format

Prior to Xcode 2.3 the Xcode debugger supported only the Stabs format. Xcode 2.3 added support for the DWARF debugging format. DWARF is a popular debugging format in the Unix world. It is more descriptive and flexible than Stabs. DWARF is also more efficient so less disk space is needed to store debugging information.

Although Xcode 2.3 added DWARF support, all Xcode projects except the Carbon C++ application projects use Stabs as the initial debugging format. To use DWARF as your debugging format:

- 1) Select your project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the Build tab.
- 4) Choose Build Options from the Collection pop-up menu.
- 5) Choose DWARF for the Debug Information Format build setting.

You have two choices for DWARF support: DWARF and DWARF with dSYM file. If you choose DWARF with dSYM file, Xcode creates a dSYM file that contains all the debugging symbols. Otherwise Xcode uses the symbols stored in the object files that were generated when Xcode compiled your program. Using a dSYM file is good for building the release version of your program. You can strip the debugging information out of the executable file to reduce its size. The debugging symbols are in the dSYM file in case you need them later.

There are some things to keep in mind if you decide to use a dSYM file. ZeroLink must be turned off. dSYM files cannot be used if you have a target that creates a static library or an object file. But you can use dSYM files for applications, frameworks, and dynamic libraries.

## Setting Debugging Options for Your Program

Xcode has options to control how Xcode debugs your program. To set debugging options for your program's executable file:

- 1) Select Executables from the Groups and Files list.
- 2) Select the name of the executable file.
- 3) Click the Info button to open the information panel for the executable file.
- 4) Click the Debugging tab to show the debugging section of the information panel.

The debugging section is where you set the options related to debugging the executable file. At the top is a pop-up menu to choose the debugger to use. Xcode has three debuggers.

- **gdb debugger.** C, C++, and Objective C programs should use the gdb debugger.
- **Java debugger,** which Java programs should use.
- **AppleScript debugger,** which AppleScript programs should use.

Xcode chooses the appropriate debugger based on the type of project you create so you shouldn't have to change debuggers. Below the When using pop-up menu is a group of options that apply to the debugger you're going to use.

## Standard Input/Output

The first option you have is choosing the program to use for standard input and output. The options are the same for debugging as they are for running your program: pseudo terminal, system console, and pipe. In most cases pseudo terminal is the best choice. With pseudo terminal Xcode's gdb console works like a Unix shell window. From the console you enter gdb debugging commands and view the results of those commands.

If you want to save your program's output to a file, choose system console. The output goes to a log file instead of the gdb console. Do not use system console if you're going to enter commands in the gdb console or if your command-line program reads input from the user. Choosing system console prevents you from typing anything in the gdb console or Xcode's run log. If you're going to perform remote debugging, choose pipe for standard input and output.

## Remote Debugging

If you're using gdb as your debugger, you can tell Xcode to debug the executable remotely. Remote debugging involves running your program on one computer and the debugger on another computer. Select the Debug execute remotely via SSH checkbox and enter the name of the computer you're connecting to in the Connect to text field. Refer to the section "Enabling Remote Debugging" in this chapter for additional information about remote debugging.

## Start Executable After Starting Debugger

The Start executable after starting debugger checkbox tells Xcode whether or not to start running your program after loading it in the debugger. If you deselect the checkbox, you must click the Restart button in the debugging window toolbar to run your program. The advantage of selecting the checkbox is that your program starts running when Xcode loads it in the debugger, which is what you normally want to happen. The advantage of not selecting the checkbox is that waiting to run your program gives you time to set breakpoints.

## Break on Debugger() and DebugStr() Calls

The Break on Debugger() and DebugStr() checkbox tells Xcode whether or not to pause your program when it reaches a call to the Debugger() and DebugStr() functions in your code. The Debugger() and DebugStr() functions are part of the Carbon framework. The Debugger() function enters the kernel debugger, and the DebugStr() function prints text to the screen. If you don't call Debugger() or DebugStr(), the checkbox is irrelevant.

## Adding Places to Look for Files

Finally, you can tell Xcode additional places to find source code files. To add a directory, click the + button and type the path name to the source code files. If you have your source code files in your project folder, you won't have to add places for Xcode to look. If you split your source code files into multiple folders and spread the folders all over your hard drive, you'll probably need to tell Xcode where to find the files.

## Setting Environment Variables for Debugging

Mac OS X's malloc library has a collection of environment variables to help debug memory allocations. Table 2.1 lists the variables. To set environment variables in Xcode:

- 1) Select Executables from the Groups and Files list.
- 2) Select the name of the executable file.
- 3) Click the Info button to open the executable settings panel.
- 4) Click the Arguments tab.
- 5) To add an environment variable, click the + button in the section Variables to be set in the environment.
- 6) Give the environment variable a name and a value. Giving an environment variable a value of zero disables the variable.

## Using the Debug Version of Frameworks

Apple provides three versions of its frameworks: standard, debug and profile. The debug version writes debugging information to Xcode's run log as your program runs. You can debug your program with all three versions of Apple's frameworks. The debug version provides more debugging information. To switch to the debug versions of Apple's frameworks:

- 1) Select Executables from the Groups and Files list.
- 2) Select the name of the executable file.
- 3) Click the Info button to open the executable settings panel.
- 4) Click the General tab.
- 5) Choose debug from the Use suffix when loading frameworks pop-up menu.

After switching to the debug version of Apple's frameworks, clean and rebuild your project for the switch to take effect.

## Guard Malloc

Memory errors in your code are difficult to debug because the problems seem to occur randomly. Your program may run fine one time and crash the next time you run it. The Guard Malloc library eliminates the randomness. When your program commits a memory access error, Guard Malloc crashes the program, which helps you locate the source of the error.

The downside of using Guard Malloc is that your program runs up to 100 times slower when running with Guard Malloc. Because of the slowdown, I recommend not using Guard Malloc the first time you debug your program. Correct the errors you find without Guard Malloc before debugging with Guard Malloc. To turn on Guard Malloc, choose Debug > Enable Guard Malloc before you start debugging your program.



Table 2.1 Malloc Library Environment Variables

Variable Name	Value	Description
<code>MallocStackLogging</code>	Any integer > 0	Logs the chain of functions your program called to make the memory allocation.
<code>MallocStackLoggingNoCompact</code>	Any integer > 0	Does what <code>MallocStackLogging</code> does and remembers call stacks of memory allocations that no longer exist. Use <code>MallocStackLoggingNoCompact</code> to remember every memory allocation made at a certain memory address.
<code>MallocScribble</code>	Any integer > 0	When your program frees memory, the operating system fills the memory with garbage values so your program can't accidentally access the memory again.
<code>MallocGuardEdges</code>	Any integer > 0	For large (over 4096 bytes) buffers the operating system places guard buffers on each edge of the buffer. If your program tries to read or write past the buffer, the program will crash.
<code>MallocDoNotProtectPrelude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer in front of the buffer.
<code>MallocDoNotProtectPostlude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer behind the buffer.
<code>MallocCheckHeapStart</code>	The number of allocations before checking the heap.	Checks the heap for corruption after $x$ memory allocations, where $x$ is the value you supply to <code>MallocCheckHeapStart</code> .
<code>MallocCheckHeapEach</code>	The interval to check the heap after the initial heap check.	Works with the <code>MallocCheckHeapStart</code> variable. After making the initial heap check with <code>MallocCheckHeapStart</code> , the operating system checks the heap every $x$ memory allocations, where $x$ is the value you supply to <code>MallocCheckHeapEach</code> .
<code>MallocCheckHeapSleep</code>	Any integer > 0	Your program goes to sleep when a heap corruption occurs.
<code>MallocCheckHeapAbort</code>	Any integer > 0	Aborts your program when a heap corruption occurs.
<code>MallocBadFreeAbort</code>	Any integer > 0	Reports an error when your program illegally frees memory.

## Enabling Remote Debugging

Remote debugging allows you to run your program on one computer while debugging it from a second computer. Fullscreen programs benefit the most from remote debugging. Without remote debugging, fullscreen programs are difficult to debug because there's no way to see the debugger's window when the program is running.

Setting up remote debugging requires a surprising amount of effort. I am going to assume you're going to be debugging on your Mac. I will refer to the computer you're going to run your program on as the host computer. You must perform the following tasks to set up remote debugging:

- The host computer must allow remote login so your Mac can connect to it.
- You must generate a key identifying your Mac and send the key to the host computer.
- You must configure your project's build folder so your Mac and the host computer can access the files in the folder.
- You must set your executable file to allow remote debugging.

## Allowing Remote Login

If you're going to debug a program running on another computer, the host computer must be configured to allow you to login to it remotely. Go to the computer you want to login to and perform the following steps:

- 1) Choose Apple > System Preferences.
- 2) Choose Sharing from the System Preferences panel.
- 3) Select the Remote Login checkbox to activate remote login.

## Generating ssh Keys

Xcode's remote debugging uses secure shell (ssh) to communicate between the two computers. You must generate a ssh key identifying your Mac so your Mac can communicate with the host computer.

Use the `ssh-keygen` command to generate ssh keys. What you supply to the `ssh-keygen` command depends on the version of ssh you're using. Most of you have version 2, the latest one. In version 2, use the following command to generate the keys:

```
ssh-keygen -t dsa
```

If you have version 1 of ssh, use the following command to generate the keys:

```
ssh-keygen -t rsa1
```

When you run the `ssh-keygen` command, ssh asks you for a file name to save the keys. ssh provides a default file in parentheses with the path `/Users/Username/.ssh/Filename`. Press the Return key to use the default file. Next, ssh asks you for a password. Be careful typing your password. The shell window provides no feedback when typing the password so it looks like you're not typing anything, even though you are typing the password.

After providing a file location and password, ssh generates two keys. The public key has the extension `.pub`, and the private key has no extension. In ssh version 2 the key name is `id_dsa`. In version 1 the key name is `identity`. Keep the private key on your computer and send the public key to the host computer.

The public key you generated goes into the file `authorized_keys2` inside the `.ssh` directory on the host computer. The `authorized_keys2` file contains the keys of all the people authorized to access the host computer. Use the `scp` command to copy the public key to the host computer. The `scp` command takes the following form:

```
scp FileName Username@MachineName
```

The machine name takes the form `Name.local`. If Emma wants to copy her key to the network server named `Server.local`, she would use the following command to copy her key:

```
scp id_dsa.pub emma@Server.local
```

After copying the public key file to the host computer, you must add the key to the `authorized_keys2` file. The `cat` command appends a file to the contents of another file. Use the `cat` command to append the public key file, `id_dsa.pub`, to the list of keys in `authorized_keys2`. After appending the public key you can remove the copy of `id_dsa.pub` on the host computer by using the `rm` command, as you can see in the following example:

```
cat id_dsa.pub > ~/.ssh/authorized_keys2
rm id_dsa.pub
```

## Creating a Shared Folder for the Project's Build Products

For remote debugging to work, both computers must be able to access the project's build folder. Either use a network directory to store the project or make the build folder a shared folder. To make a folder a shared folder:

- 1) Choose Apple > System Preferences.
- 2) Choose Sharing from the System Preferences panel.
- 3) Select the Personal File Sharing checkbox to make the folder a shared folder.

After setting up the build folder to let both computers access them, you must tell Xcode where the build products and intermediate files are.

- 1) Select your project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the Build tab. Some versions of Xcode have a Styles tab instead of a Build tab.
- 4) Choose Build Locations from the Collections pop-up menu.
- 5) Set the locations for the build products path and the intermediate files path to the shared folder.

## Turning on Remote Debugging

The final step to enabling remote debugging is to turn it on for your program. To turn on remote debugging:

- 1) Select Executables from the Group and Files list.
- 2) Select the name of the executable file.
- 3) Click the Info button to open the information panel.
- 4) Click the Debugging tab.
- 5) Select the Debug executable remotely via SSH checkbox.
- 6) Enter the name of the computer on which you want to run your program in the Connect to text field.

## Launching the Debugger

The setup work is done. Let the debugging begin. Xcode provides several ways to launch the debugger.

- Choose **Debugger > Show Debugger** to open the debugger window. Launch your program by clicking the **Debug** button on the debugger window toolbar.
- Choose **Build > Build and Debug** to build your program, open the debugger window, and launch your program.
- If you have built your program, choose **Debug > Debug Executable** to open the debugger window and launch your program.
- Xcode 2.2 lets you attach running programs to the debugger. Choose **Debug > Attach > Program Name**.

Figure 2.1 shows the debugger window. The window has four sections.

- **Toolbar**, which has buttons for the most common debugging commands.
- **Call stack viewer**.
- **Variable viewer**.
- **Editor**, which is identical to the editor you use to enter your source code.

## Call Stack Viewer

The call stack viewer lets you view your program's call stack, which is the chain of function calls that led your program to its current location. Selecting a function from the list takes you to its location in the editor, assuming you wrote that function. Sometimes low-level system calls appear in the call stack; selecting them won't do anything.

The call stack viewer has a pop-up menu that lets you select a thread to display in the viewer. Normally your code will have two threads, one for your program and one for the debugger. In this case Xcode displays the call stack for your program. If you write code with multiple threads, use the thread pop-up menu to examine your program's threads.

## Variable Viewer

Use the variable viewer to look at your program's variables. It initially has three columns of information for each variable.

- The variable name.
- The variable's value.
- A summary, which provides additional information about the variable. If you have a variable of type `Rect`, the Summary column tells you the values of the rectangle's four corners. Many variables have an empty Summary column.



**Figure 2.1**

Xcode debugger window.

Choosing Debug > Variables View > Show Types adds a fourth column to the variable viewer. This column shows the variable's data type.

Xcode breaks variables down into the following categories:

- Arguments, which contains any arguments your program passed to the current function.
- Locals, which contains the function's local variables.
- File Statics, which contains any static variables in the source code file. Constants are a major source of static variables.
- Globals, which contains your program's global variables.
- Registers, which contains the contents of your Mac's registers.

If a variable has multiple fields of information, such as C structs, C++ classes, Objective C classes, or arrays, it will have a disclosure triangle next to it. Click the disclosure triangle to expand the variable to see the other fields. These fields may in turn have more fields; you can end up doing a lot of triangle clicking.

When you're debugging you normally have a set of variables you're most interested in viewing. You can place these variables in a separate window so you don't have to click as many disclosure triangles. Select the variables you want in the debugger window and choose Debug > Variables View > View Variable in Window.

Double-clicking a variable's value lets you edit the value by typing in a new one. Suppose you want to test your program's handling of null pointers. By setting a pointer variable to `NULL` in the variable viewer, you force your null pointer handling code to execute.

## Custom Data Formatters

Custom data formatters let you customize what appears in the Value and Summary columns for each variable. Data formatters are especially useful for displaying the data structures you create for your programs. By using data formatters, you can display the most important data structure information in the Summary column. For data formatters to work, they must be enabled in Xcode. Choose Debug > Variables View and make sure the Enable Data Formatters menu item has a check mark next to it.

To create a custom data formatter, double-click the Value or Summary column for a variable and enter a format string. Any literal text you enter will appear in the debugger window exactly as you type it, which makes the literal text good for labels. To refer to the variable itself, use the value `$VAR`. If the variable is a data structure, you can refer to the structure's individual members by placing the character `%` before and after the member name.

```
%MemberName%
```

If one of your data structure's members is another data structure, use the dot operator. Suppose you have a data structure with a member named `area`, which is of type `Rect`. You want to show the left edge in the format string. You would type the following format string:

```
%area.left%
```

Let's walk through a simple example of using data formatters. Suppose you have a data structure for 3D vectors named `Vector3D` with members `x`, `y`, and `z`. You would like the Summary column to show the `x`, `y`, and `z` components of the vector. Select the `Vector3D` variable in the debugger window and double-click the Summary column. Enter the following text in the Summary column:

```
x=%x%, y=%y%, z=%z%
```

Now every `Vector3D` variable in your program will show the `x`, `y`, and `z` components in the Summary column. If `x` has a value of 1, `y` has a value of 3, and `z` has a value of 5, the Summary column shows the following output:

```
x=1, y=3, z=5
```

## Viewing Shared Libraries

Xcode 2.2 lets you see which libraries have been loaded by your program. Choose **Debug > Tools > Shared Libraries** to open the Shared Libraries window, which you can see in Figure 2.2.

At the top of the window are two pop-up menus that let you specify the default level of debugging symbols to show for system and user libraries. Most of the shared libraries in your programs are going to be system libraries. Only libraries that your project's targets produce are user libraries.

There are three levels of symbol showing. You can tell the debugger to show all symbols, no symbols, or external symbols, symbols declared external in the library's code. Showing external symbols is the default. The debugger loads all external symbols and loads other symbols when necessary.

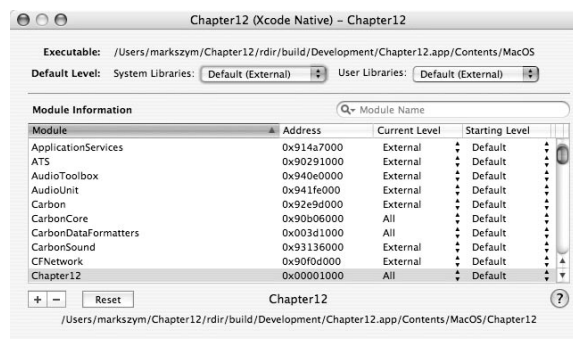
Why so much concern over symbol levels? Can't you show all debugging symbols and be done with it? Showing debugging symbols is a balance between the ability to debug and speed of debugging. Mac OS X applications use a lot of shared libraries, and these libraries have lots of symbols. Loading every one of these symbols takes time and make debugging slower. Loading external symbols balances your need for information with your need for speed.

Below the pop-up menus is a list of libraries. The Shared Libraries window displays the following information for each library:

- The name of the library.
- The library's memory address. If there is no memory address, the library has not been loaded.
- The current level of debugging symbols to display for the library.
- The starting level of debugging symbols to display for the library.

Each library has a menu to set the current and starting level of debugging symbols. If you have some libraries where you want to see more debugging symbols, you can set the starting level for those libraries. Setting the current level is useful if you want to see more debugging symbols at certain times when you're debugging.

Clicking the Reset button sets the starting level of debugging symbols back to the default value you set in the pop-up menus. Select a library and click the minus button to remove a library from the Shared Libraries window. Click the + button to add a library to the window.



**Figure 2.2**

Shared libraries window..

## Viewing Global Variables

Xcode does not automatically display the contents of global variables, which can be a problem if you use global variables in your programs. You must tell Xcode the global variables it should show in the variable viewer. Choose Debug > Tools > Global Variables to open the global variables browser, which you can see in Figure 2.3. In Xcode 2.2, clicking the disclosure triangle next to Globals in the variable viewer also opens the global variables browser.

The left side of the global variables browser contains all the libraries in your program. Selecting a library from the list fills the right side of the window with the library's global variables. Select the checkbox next to a global variable to tell Xcode to display it in the variable viewer.

## Viewing Registers

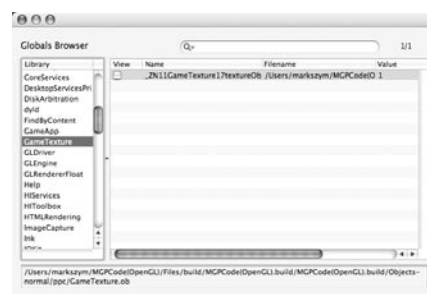
To look at the contents of the computer's registers, click the disclosure triangle next to Registers in the variable viewer. You must be viewing disassembled code to be able to view registers with Xcode 2.2. Choose Debug > Toggle Disassembly Display to see disassembled code in the debugger window. Registers are memory units in the CPU. The CPU uses registers as operands for assembly language instructions. Plus, registers store instructions and the results of instructions. If your program adds two numbers, the CPU uses three registers: two for the two numbers you're adding and one to store the sum. The lower the level of code you write, the more likely you'll be to view registers.

Table 2.2 lists the registers Xcode displays for PowerPC Macs. The book *PowerPC Microprocessor Family: The Programming Environments* contains detailed information about the PowerPC's registers. You can download the book from Motorola or IBM's PowerPC websites. Table 2.3 lists the registers Xcode displays for Intel Macs. The *IA-32 Intel Architecture Software Developer's Manual* contains detailed information about the Intel processor's registers. You can download the book from Intel's website.

# Breakpoints

To get any real debugging done, you need to use breakpoints. Technically, you can click the Pause button in the debugger window then step through your code. But Mac OS X programs with a GUI spend most of their time waiting for user input. When waiting for input your program isn't doing anything so there's nothing to debug.

Breakpoints tell the debugger to pause execution of your program at a specific line of code in your program. From there you can walk through the code line by line to find out what's wrong. By using breakpoints you can focus on a small section of code.



**Figure 2.3**

Global variables browser.

Table 2.2 PowerPC Registers

Register	Description
r0–r31	Integer registers.
f0–f31	Floating-point registers.
v0–v31	Vector registers. These registers can store multiple values in one register, letting you perform multiple operations with one machine instruction. Macs with a G3 processor do not have vector registers, but all other Macs capable of running OS X have them.
pc	Program counter, which the CPU uses to keep track of where it is in your program.
ps	Machine state register, which stores the current state of the CPU. If you have a G5 Mac, one of the things the machine state register stores is whether the CPU is running in 64-bit or 32-bit mode.
cr	Condition register, which handles testing conditions and branching to other parts of your program.
lr	Link register, which contains a target address for branch instructions. When your program calls a function, the CPU saves the return address, where the program should go when the called function finishes, in the link register.
ctr	Count register, which can contain a loop count the computer can decrement or contain a target address for a branch instruction. Loops use the count register.
xer	Integer exception register, which indicates overflow and carry conditions for math operations.
mq	Multiply quotient register, which the computer uses for multiplying and dividing. Only PowerPC 601 chips have the multiply quotient register so Mac OS X programs don't have to worry about this register.
fpscr	Floating-point status and control register, which deals with exceptions and rounding.
vscr	Vector status and control register, which currently tells you whether floating-point instructions perform in a Java-compliant matter and tells you about overflow for math operations.
vrsave	Vector save/restore register, which helps programs save and restore the architectural state.

To set a breakpoint open the source code file and click the gutter next to the line of code where you want the breakpoint. The gutter has a gray arrow when a breakpoint is set, as you can see in Figure 2.4. Clicking the gray arrow disables the breakpoint. Drag the arrow off the gutter to remove the breakpoint permanently.

To see all the breakpoints you've set in your program, click the Breakpoints button in the debugging window toolbar. Clicking the Breakpoints button opens the breakpoints window, which shows the breakpoints. You can use the breakpoints window to remove a breakpoint by selecting the breakpoint and choosing Edit > Delete.

Xcode 2.1 brought a lot of changes to the breakpoints window. I've decided to create separate sections for the old and new versions of the breakpoints window.



Figure 2.4

Setting a breakpoint.



**Table 2.3 Intel Registers**

Register	Description
EAX	Accumulator for arithmetic operations.
EBX	Pointer to data in the DS segment.
ECX	Counter for loops and string operations.
EDX	I/O pointer.
ESI	Source pointer for string operations. It can also be used as a pointer to data the DS register points to.
EDI	Destination pointer for string operations. It can also be used as a pointer to data the ES register points to.
EBP	Pointer to data on the stack.
ESP	Stack pointer.
CS	Code segment register. The segment registers help the CPU translate logical memory addresses to linear addresses. The CPU then translates the linear addresses to physical addresses.
DS	Data segment register.
SS	Stack segment register.
ES	Data segment register.
FS	Data segment register.
GS	Data segment register.
EFLAGS	Contains a group of status flags.
EIP	Instruction pointer. It contains the location of the next instruction to be executed.
MM0-MM7	Registers used with Intel's MMX technology. MMX registers are 64 bits.
XMM0-XMM7	Registers used with Streaming SIMD Extensions (SSE). SSE is similar to AltiVec on PowerPC processors. SSE registers are 128 bits.
MXSCR	Status and control register for SSE operations.

## The Breakpoints Window Before Xcode 2.1

The breakpoints window for older versions of Xcode is basic. It lists each breakpoint you've set. Next to each breakpoint is a checkbox. If the checkbox is selected, the breakpoint is enabled, which means Xcode pauses your program when it hits the breakpoint.

Deselecting the checkbox next to a breakpoint in the breakpoints window disables the breakpoint. Disabling a breakpoint keeps it in the breakpoint list, but the debugger won't stop at disabled breakpoints. Why would you want to disable a breakpoint? Suppose you set a breakpoint inside a loop that executes 50 times. You may want to step through the code only the first time through the loop. In this case you would set the breakpoint, step through the code, then disable the breakpoint. To restore the breakpoint select the checkbox next to the breakpoint.

Clicking the New Symbolic Breakpoint button adds a symbolic breakpoint. Symbolic breakpoints pause your program when it enters a function. After clicking the New Symbolic Breakpoint button, enter the name of the function where you want to pause. The debugger will pause your program's execution when it reaches that function in your program.

## The Breakpoints Window After Xcode 2.1

The only similarity Xcode 2.1's breakpoints window has with previous versions is that they both list your program's breakpoints. Figure 2.5 shows Xcode's 2.1's breakpoints window. For each breakpoint in your program, the breakpoint window lists the following information:

- An icon identifying the type of breakpoint: symbolic or line.
- The name of the breakpoint.
- A checkbox that tells you if the breakpoint is enabled.
- A condition that tells Xcode when to pause your program. A blank condition tells Xcode to pause your program every time it reaches the breakpoint.
- A checkbox that tells Xcode to go back to your program after executing an action.

In the breakpoints list is an entry with the name Double-Click For Symbol. Double-clicking that entry and typing in a function name adds a symbolic breakpoint.

Each breakpoint in the breakpoints window has a disclosure triangle next to it. Click the disclosure triangle to tell Xcode to perform an action when it reaches the breakpoint. There are six types of actions to perform.

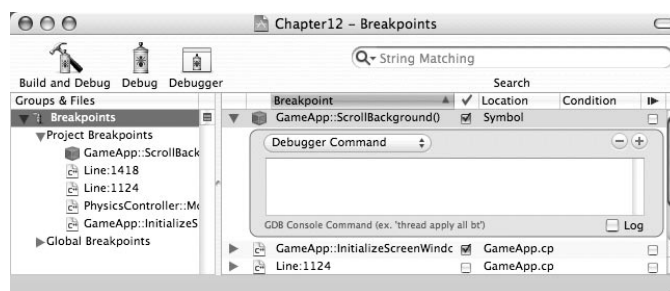
- Debugger commands execute `gdb` commands. Refer to the section “Using the `gdb` Console” later in the chapter to learn about the most common `gdb` commands.
- Log writes a message to the console.
- Sound plays a sound.
- Visualize shows the class model.
- Shell command executes a Unix shell command. Use a shell command to run shell scripts.
- AppleScript executes an AppleScript.

## Setting Watchpoints in Xcode

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable.

If you're using a version of Xcode older than 2.1, you must use the `gdb` console to set watchpoints. Refer to the section “Setting Watchpoints” later in the chapter for more information.

For those of you running Xcode 2.1, setting a watchpoint is easier. Control-click the variable you want to set a watchpoint for. Choose Watch Variable from the contextual menu that opens.



**Figure 2.5**

Xcode's  
breakpoints  
window.

## Stepping Through Your Code

When debugging you will spend a lot of time stepping through your code. Stepping means executing one line of code at a time, letting you pinpoint problems in your program.

The debugging toolbar has three buttons for stepping: Step Over, Step Into, and Step Out. To demonstrate stepping I will use a small C function.

```
void ExampleFunction(void)
{
1   int routinesCalled;

2   routinesCalled = 0;
3   FirstRoutine();
4   routinesCalled++;

5   SecondRoutine();
6   routinesCalled++;
}
```

Assume you've paused the program at `ExampleFunction()`. The debugger moves past line 1 and stops at line 2. For this line of code, the Step Over and Step Into functions do the same thing. They execute the line of code, giving `routinesCalled` the value zero, and move to line 3.

If you click the Step Over button again, the debugger executes all the code in the function `FirstRoutine()` and moves to line 4. The debugger *steps over* the function call and moves to the next line of code. Clicking Step Over a second time moves the debugger to line 5.

Clicking the Step Into button at this point moves the debugger inside `SecondRoutine()`. From there you can step through the code in `SecondRoutine()`, which I have not bothered to list. The debugger *steps into* the function you're calling. After executing all the lines of code in `SecondRoutine()`, the debugger moves to line 6. If you don't want to walk through every line of code in `SecondRoutine()`, click the Step Out button when you're finished looking at `SecondRoutine()`. The debugger will *step out* of `SecondRoutine()` and take you back to `ExampleFunction()`.

On a line of code that calls another function, clicking Step Over executes the code in the function. Clicking Step Into takes you inside the called function. Clicking Step Out takes you out of the called function.

## Viewing Memory

To look at the contents of memory, open the memory browser by choosing Debug > Tools > Memory Browser. The memory browser, shown in Figure 2.6, has two combo boxes and two pop-up menus to control what appears in the memory browser and control how the memory appears. The Address combo box lets you choose the starting memory address to appear in the browser. You can enter an address in the text field, choose from a list of previously viewed addresses, or use the little arrows to choose the starting address. Selecting a variable from the debugger window and choosing Debug > Variables > View As Memory displays the variable's memory contents in the memory browser.

The Bytes combo box determines the amount of memory the browser displays. The combo box has initial values ranging from 1 KB to 1 MB. You can also type in the amount of bytes to show.

The Word Size pop-up menu determines how many bytes of memory appear in one column of the browser. The possible word sizes are 1, 2, 4, and 8 bytes. The Columns pop-up menu determines how many columns appear in one row of the browser. The browser can display 1, 2, 4, 8, 16, or 32 columns.

Each row in the memory browser displays the following information:

- The starting address for the row.
- The contents of memory, displayed as hexadecimal numbers.
- The ASCII character corresponding to the value of each byte of memory in the row.

Xcode 2.2 added the Auto Update checkbox to the memory browser. If this checkbox is selected, the memory browser updates its contents when you step through code.

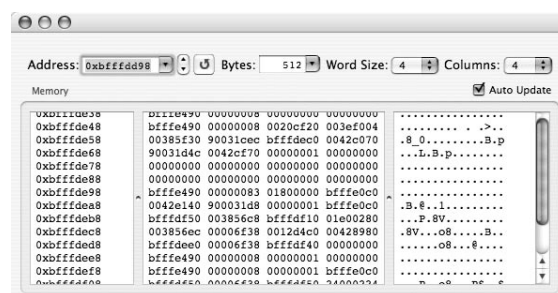
## Looking at Disassembled Code

Choosing Debug > Toggle Disassembly Display splits the editor pane. The left pane contains the original version of your code. The right pane shows the corresponding assembly language instructions. Showing the disassembled code lets you see the assembly language instructions your program executes as you step through your program. To examine the contents of registers on Xcode 2.2, you must look at disassembled code.

## Fixing Your Code While Debugging

The purpose of debugging is to find the source of your program's errors so you can correct them. With Xcode you can correct errors without leaving the debugger by using Fix and Continue. Make your changes in the editor and click the Fix button in the toolbar. A dialog box opens, asking if you want to save the changes you made to the file. Save them or else Fix and Continue won't work. Xcode compiles the file you modified and resumes execution where you last stopped it.

When you make a correction to your code, you want the debugger to stop where you made the correction so you can make sure your fix worked. The line of code you fixed may not be where the program counter, the current line of code, is. You can manually manipulate the program counter to make Xcode resume your program where you want after clicking the Fix button. The program counter has a red arrow in the gutter, and Xcode highlights the line of code in red, which you can see in Figure 2.7. You may have a hard time seeing the arrow if you set a breakpoint at that line of code. Drag the arrow to the line of code where you want Xcode to stop after compiling your corrections. This line of code will now be highlighted in red. Click the Fix button, and Xcode stops where you specified after recompiling the program.



**Figure 2.6**

Xcode's memory browser.

Fix and Continue works on only one file at a time. If you find errors in two source code files, you must correct the errors in one file, click Fix, correct the errors in the second file, and click Fix again. Fix and Continue working on one file at a time is not as big a limitation as you might think. You should fix one error at a time, and one error rarely spreads across multiple files.

## Debugging Command-Line Programs

For the most part debugging a command-line program is no different than debugging a GUI program. But command-line programs present one challenge. They require a console to enter input and display output. How do you enter input and display output when debugging a command-line program in Xcode?

The answer is to use Xcode's standard I/O log, which provides the console you need to enter input and display output. Choose Debug > Standard I/O Log to open the log window. You must start debugging your program before you can open the standard I/O log. Your program's output will appear in the log, and you will be able to enter any input your program needs.

## Using the gdb Console

The Xcode debugger handles basic debugging tasks; you can step through your code line by line, examine the values of variables, and set breakpoints. Many of you will be able to do all your debugging from the Xcode GUI. But if you want to do anything more advanced, you must use the gdb console and type commands in it. Xcode uses the gdb debugger for C, C++, and Objective C programs.

### NOTE

You cannot use the gdb console for Java and AppleScript programs because neither language uses gdb for debugging. Java programs use the Java debugger and AppleScript programs use the AppleScript debugger.

Using the console doesn't mean you have to abandon the Xcode GUI. You can use the debugger window toolbar to step through your code and view your program's variables in the debugger window while using the console to perform tasks the Xcode GUI can't handle.

Displaying the console is easy in Xcode. Click the Console button in the debugger window toolbar. Before you can type commands in the console, you must pause your program's execution in the debugger. If you've set breakpoints in your program, you can wait for your program to reach one of them. Otherwise you can click the Pause button on the debugger window toolbar to start entering commands in the gdb console.



**Figure 2.7**

Program counter.

## Stopping Program Execution

There are three ways to stop your program's execution without explicitly pausing your program: breakpoints, watchpoints, and catchpoints. As I mentioned earlier in this chapter, breakpoints stop your program's execution at a specific line of code in your program. You can set breakpoints in the `gdb` console the way you would from the Xcode GUI, but `gdb` provides even more options. From the console you can set conditional breakpoints, which take effect only if it meets a condition you specify, and set breakpoints that execute only one time.

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable.

Catchpoints stop your program's execution when a C++ exception occurs. Exceptions provide a way of handling run-time errors in C++ programs. The C++ statements `try`, `catch`, and `throw` deal with exceptions. Obviously, only C++ programs can use catchpoints.

## Setting Breakpoints

The `break` command, which you can abbreviate by typing `b`, sets breakpoints in the console. There are many ways to set them in `gdb`. If you supply no arguments.

```
break
```

`gdb` sets a breakpoint at the current line of code. Supplying a line number tells `gdb` to set a breakpoint at that line number in the current source code file. The following command:

```
break 447
```

Tells `gdb` to set a breakpoint at line 447 in the current file. To set a breakpoint at a specific line in a source code file, supply the file name, a colon, and the line number. The following command:

```
break main.m:15
```

Tells `gdb` to set a breakpoint at line 15 in the file `main.m`. Rather than setting a breakpoint at a line of code, you may prefer to set a breakpoint at the start of a function. Supply the function name when calling `break`. If you had a C function `PlayMusic()` that you wanted to set a breakpoint at, you would enter the following command:

```
break PlayMusic
```

Object-oriented languages like C++ and Objective C complicate matters slightly. You must also supply a class name along with the function name because multiple classes can have a function with the same name. If you had a C++ class named `TMusic` with a member function `Play()`, you would enter the following command:

```
break TMusic::Play
```

If you had an Objective C class named `TMusic` with a `Play:` method, you would enter the following command:

```
break -[TMusic Play:]
```

To set a breakpoint that stops your program only once, call `tbreak`. The `t` in `tbreak` stands for temporary. The `tbreak` command takes the same arguments as `break` does. The following command:

```
tbreak main.m:15
```

Sets a breakpoint at line 15 in the file `main.m` then deletes the breakpoint when your program stops at it.

## Setting Watchpoints

The `watch` command sets watchpoints in your program. You supply an expression, which in most cases is a variable name. When the expression's value changes, `gdb` pauses your program.

```
watch x
```

## Setting Catchpoints

The `catch` command sets catchpoints in your program. There are two ways to call `catch`.

```
catch catch
```

```
catch throw
```

The first statement stops your program when it catches a C++ exception, and the second statement stops your program when it throws a C++ exception.

## Examining Your Breakpoints

To view a list of all the breakpoints, watchpoints, and catchpoints, use the `info breakpoints` command. For each breakpoint, watchpoint, and catchpoint, `gdb` displays the information shown in Table 2.4.

**Table 2.4 Information `info breakpoints` Provides**

Information	Description
Breakpoint number	Breakpoint numbers start with 1. Many breakpoint-related <code>gdb</code> commands take breakpoint numbers as parameters.
Type	Tells you whether you have a breakpoint, watchpoint, or catchpoint.
Disposition	Specifies what should happen when your program hits the breakpoint. The value <code>keep</code> tells <code>gdb</code> to keep the breakpoint. The value <code>dis</code> tells <code>gdb</code> to disable the breakpoint, and the value <code>del</code> tells <code>gdb</code> to delete the breakpoint.
Enabled	Is the breakpoint enabled? If so, it will have the value <code>y</code> . If not, it will have the value <code>n</code> .
Address	The memory address of the breakpoint.
What	The function, source code file, and line number of the breakpoint
Condition	If the breakpoint is conditional, the condition will reside here. If the breakpoint has been hit before, the number of times it's been hit will be here as well.

## Setting Conditional Breakpoints

Conditional breakpoints give you the power to control when a breakpoint pauses your program. Supply a condition when you set the breakpoint. When your program hits the line of code where you set the conditional breakpoint, gdb pauses the program if the condition you supply is true.

There are two ways to set conditional breakpoints. First, supply a condition when you create the breakpoint with the `break if` command. The condition can be any Boolean expression, an expression that can be either true or false. Suppose you had the following code in a program:

```
enum DirectionType {  
    NORTH, SOUTH, EAST, WEST  
};  
  
void Move(DirectionType directionToMove);
```

The following breakpoint executes when the program moves north:

```
break Move if directionToMove == NORTH
```

You can also use `if` to set conditional watchpoints, but you won't use conditional watchpoints as much as conditional breakpoints. Watchpoints pause your program when an expression changes value. The changing of value is what you're normally interested in so regular watchpoints are usually sufficient. Use a conditional watchpoint to pause your program when an expression changes to an interesting value.

```
watch x if x > 50
```

Second, use the `condition` command to turn a regular breakpoint into a conditional one. With the `condition` command you can set your breakpoints in Xcode and use the console to make certain ones conditional. Supply a breakpoint number and a condition with the `condition` command. The example below shows how you would use the `condition` command to pause when the `Move()` function moves north, assuming the breakpoint number is 1.

```
condition 1 directionToMove == NORTH
```

You can also use the `condition` command to make watchpoints and catchpoints conditional. Use conditional catchpoints to pause your program when it raises a specific exception.

To make a conditional breakpoint, watchpoint, or catchpoint unconditional, use the `condition` command. Supply a breakpoint number and no condition, and gdb removes the condition from the breakpoint, as you can see in the example below.

```
condition 1
```

Closely related to conditional breakpoints is gdb's ignore count. The ignore count lets you skip a breakpoint when your program reaches it. Normally the ignore count is zero, meaning you won't skip any breakpoints, but you can use the `ignore` command to ignore a breakpoint. You supply a breakpoint number and a desired ignore count, and gdb will ignore that breakpoint the number of times you specify. The following example skips breakpoint 2 seven times:

```
ignore 2 7
```



When you reach a breakpoint in your program, you can set the ignore count for that breakpoint when you resume if you use the `continue` command from the console instead of clicking the Continue button in the Xcode debugger window. The example below sets the ignore count to nine.

```
continue 9
```

## Disabling and Deleting Breakpoints

Let's say you set a breakpoint inside a loop. If the loop runs hundreds of times, you may not want to step through your code every time through the loop. Disabling the breakpoint turns it off but keeps it in the breakpoints list so you can turn it back on later.

The `disable` command disables breakpoints, watchpoints, and catchpoints. You supply a breakpoint number or a range of breakpoint numbers. The `info breakpoints` command shows your program's breakpoint numbers.

```
disable 4  
  
disable 3-6
```

To enable disabled breakpoints, watchpoints, and catchpoints, use the `enable` command. Like the `disable` command, you supply either a breakpoint number or a range of numbers.

```
enable 2  
  
enable 8-13
```

Calling `enable` like I did in the previous example causes the breakpoint to execute every time your program hits the breakpoint. Calling `enable` with the `once` option disables the breakpoint after it stops your program once.

```
enable once 1  
  
enable once 5-6
```

Calling `enable` with the `delete` option deletes the breakpoint after it stops your program once.

```
enable delete 9  
  
enable delete 2-7
```

To delete a breakpoint, watchpoint, or catchpoint, use the `delete` command. Like the `enable` and `disable` commands, you can supply a breakpoint number or a range of numbers.

```
delete 16  
  
delete 11-14
```

If you want to delete the breakpoint you've just reached, use the `clear` command. Supply no arguments to delete the breakpoint you reached. You can also supply a line number or a function where you want to delete the breakpoint.

```

clear                // Clear the current breakpoint

clear main.m:15      // Clear a breakpoint at a line

clear PlayMusic      // Clear a C function breakpoint

clear TMusic::Play    // Clear a C++ member function breakpoint

clear -[TMusic Play:] // Clear an Objective C method breakpoint

```

## Command Lists

Command lists allow you to run a series of commands when the debugger hits a breakpoint, watchpoint, or catchpoint. A command list adheres to the following format:

```

commands [Breakpoint Number]
    Insert the commands you want to execute here
end

```

If you create the command list immediately after creating the breakpoint (or watchpoint or catchpoint) you do not have to supply a breakpoint number; `gdb` attaches the command list to the breakpoint you just created. Use the `info breakpoints` command to get the numbers of all your program's breakpoints, watchpoints, and catchpoints.

When you start building a command list by typing commands, the prompt changes from `(gdb)` to `>` and will not revert back to `(gdb)` until you finish building the list by typing `end`. Many command lists have `silent` as the first command. The `silent` command suppresses the printing of the message that you hit a breakpoint, which helps if your command list prints information to the console. For the `silent` command to work, it must be the first command in a command list.

You can execute as many commands as you want in a command list, and you can use any `gdb` command in the list. The following command list writes the contents of an integer variable `y` to the console and continues the program's execution when it reaches the first breakpoint in the program:

```

commands 1
    silent
    printf "y = %d \n", y
    continue
end

```

You can also specify a command list when you create a breakpoint. The example below demonstrates creating a command list after creating a conditional breakpoint.

```

break main.m:24 if y > 5
commands
    silent
    printf "y = %d \n", y
    continue
end

```

**NOTE**

You don't have to indent the commands in the gdb console. I indented the examples to make them easier to read.

**Examining Data**

Xcode's debugger window lets you view your variables and their values. The memory browser lets you view the contents of memory. From the gdb console you can view dynamic arrays. You can also tell the console to show the values of variables automatically when your program stops execution.

**Examining Dynamic Arrays**

Many programs use pointers to create arrays when the size cannot be determined until the program runs. To create these dynamic arrays, you declare a pointer variable, then use the `malloc()` (in C) or `new()` (in C++) calls to make the pointer large enough to store the array. When you look at the pointer variable, you want to look at the array, not the variable. If you look at the pointer variable in the variable viewer of Xcode's debugging window, you'll see only the pointer, not the array.

Artificial arrays solve the pointer problem. The `@` operator defines the array. The array goes to the left of the `@` operator, and the length of the array goes to the right of the array. Use the `print` command to look at the contents of the array.

```
print *array@length
```

The length is the number of elements you want to display. Suppose you have a 1000 element array, `myArray`, and you want to look at the first 50 elements. You would view the 50 elements with the following command:

```
print *myArray@50
```

If you want to start looking from a position other than the start of the array, use the following notation:

```
print *(array + starting point)@length
```

If you want to skip the first 100 elements and look at the next 25, you would run the following command:

```
print *(myArray + 100)@25
```

**Displaying Data Automatically**

When you're debugging, there's a few variables you're especially interested in. gdb allows you to place these variables in a display list, which gdb displays every time your program stops. Display lists keep you from having to click a series of disclosure triangles in the variable viewer to look at an important variable.

To add a variable to the display list, use the `display` command. Supply a variable name, and gdb adds it to the list.

```
display x
```

You can also supply any of the viewing formats in Table 2.5 to tell `gdb` to display the data the way you want. The following example displays a variable as a floating-point number:

```
display/f y
```

To see a list of all the variables you're automatically displaying, use the `info display` command. It tells you the following pieces of information for each variable:

- The ID number, which you can use to turn off automatic display.
- Whether or not it's enabled, `y` for yes and `n` for no.
- The expression to display.

To temporarily disable a variable from the automatic display list, use the `disable display` command. You supply ID numbers (separated by spaces) to disable, which you can see in the following examples:

```
disable display 5           // Disable one variable

disable display 1 3 6       // Disable multiple variables
```

The `enable display` command enables variables you disabled with the `disable display` command. Supply ID numbers to enable. The examples below enable the variables I disabled in the previous example.

```
enable display 5

enable display 1 3 6
```

To permanently remove variables from the automatic display list, use either the `undisplay` or `delete display` commands. Like the `enable display` and `disable display` commands, you supply a list of ID numbers separated by spaces, which you can see in the examples below.

```
undisplay 3

undisplay 2 4 5 9

delete display 7

delete display 2 6 13
```

**Table 2.5 Data Viewing Formats**

Format	Description
x	Display as a hexadecimal (base 16) integer. This is the default for viewing memory.
d	Display as a signed decimal integer.
u	Display as an unsigned decimal integer.
o	Display as an octal (base 8) integer.
t	Display as a binary (base 2) integer.
a	Display as hexadecimal address and as an offset from the nearest symbol.
c	Display as a character constant.
f	Display as a floating-point number.

You can place memory locations in automatic display lists so you can look at important memory addresses every time your program stops. Use the `display` command, supplying the memory address. Table 2.6 lists the memory units you can use to display memory. The following are some examples of using display lists:

```
display/64b myPtrVariable    // Display 64 bytes automatically
display/16i 0x23459900      // Display 16 machine instructions
display/100fg 0x12123400    // 100 giant words as floating-point
```

## Executing Shell Commands

The `shell` command lets you execute a Unix shell command in `gdb`. It takes the following form:

```
shell Command
```

With the `shell` command you can run another program in the `gdb` console. The following command runs the `heap` program that shows your program's memory heap:

```
shell heap ProcessID
```

Run the program Activity Monitor to find your program's process ID. You won't know the process ID until your program launches.

The `shell` command also lets you run shell scripts, giving you enormous power if you know how to write Unix shell scripts. You can also place `shell` commands in a command list to execute scripts of shell commands when your program hits a breakpoint.

## Defining Your Own Commands

If you find yourself entering a sequence of commands repeatedly, define a command to save yourself some typing. A user-defined command consists of a series of `gdb` commands. User-defined commands are similar to command lists. The difference is command lists execute when your program reaches a breakpoint while user-defined commands execute when you submit the command.

**Table 2.6 Memory Units**

Unit	Letter you pass to gdb	Number of Bytes
Bytes	b	1
Halfwords	h	2
Words	w	4
Giant words	g	8
Machine Instructions	i	N/A
String	s	N/A

## 118 Chapter 2: Debugging with Xcode

The first command in a user-defined command is the `define` command, which gives the command a name. The `define` command takes the following form:

```
define CommandName
```

After entering the `define` command, the console prompts you to enter the `gdb` commands you want to place in the user-defined command. A user-defined command can have up to ten arguments, with the first argument having the name `$arg0` and the tenth argument having the name `$arg9`. The `end` command is the equivalent of `}` in a C or Java program. Use the `end` command to end your user-defined command and return the prompt in the console to normal.

Let's look at an example of a simple user-defined command. Suppose you want to create a command that prints the contents of a dynamic array. You would define the following command:

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end
```

In this example `$arg0` is the pointer to the array, `$arg1` is the first element you want to display, and `$arg2` is the number of elements to display. To display the first 20 elements of an array called `myArray`, you would call the `PrintArray` command you created.

```
PrintArray myArray 0 20
```

`gdb` cannot alert you to errors in your user-defined command as you're entering the `gdb` commands. Running the command is the only way to know you wrote the command correctly. You have the responsibility of making sure you entered the `gdb` commands correctly.

### Conditional Commands

You may want some of the `gdb` commands in your user-defined command to execute multiple times and other commands to execute conditionally. `gdb` has `if` and `while` commands that work similarly to `if` and `while` statements in C. Use the `if` and `while` commands to create conditional commands in your user-defined command.

The `if` command executes a series of `gdb` commands if a condition is true. It takes the following form:

```
if Condition
    Commands
else
    Commands
end
```

To make the `PrintArray` command example check for a negative starting point and for a negative number of items, you would use the following command:

```
define PrintArray
    if ($arg1 < 0) || ($arg2 < 0)
        printf "ERROR! \n"
    else
        print *($arg0 + $arg1)@$arg2
    end
end
```

The `while` command executes a series of `gdb` commands repeatedly until a condition is false. It takes the following form:

```
while Condition
  Commands
end
```

The example below prints the first  $x$  elements of a static array one element at a time, where  $x$  is a number you supply.

```
define PrintStaticArray
  set $array = $arg0
  set $index = 0
  while ($index < $arg1)
    print $array[$index]
    printf("\n")
    set $index = $index + 1
  end
end
```

The variables `$array` and `$index` are convenience variables. Convenience variables store values in `gdb` so you can use the values later in the debugging session. They have no effect on your program. In the `PrintStaticArray` example the `$index` convenience variable keeps track of where `gdb` currently is in the array. Convenience variables start with `$`. The name of the convenience variable cannot be a number.

```
$3
```

`gdb` uses variable names with numbers as a value history of variables you print to the console. When you print a variable, `gdb` creates a value history variable and displays the value history variable in the console. You can refer to value history variables in other `gdb` commands.

## Documenting Your Commands

If you create commands for other programmers to use, you want to document the commands so the other programmers know what the command does. The `show user` command displays the `gdb` commands that make up a user-defined command. If you supply no arguments, the `show user` command displays the `gdb` commands for all user-defined commands. Supply a command name to show the `gdb` commands for a single user-defined command.

```
show user CommandName
```

To create more documentation about a command, use the `document` command. It takes the following form:

```
document CommandName
```

After executing the `document` command, the console prompts you to enter the documentation. Press the Return key to end a paragraph of documentation. When you're finished entering the documentation, press the Return key and type `end`. Now when someone runs `help` on your command, the documentation you entered appears in the console.

## Reading Commands from a File

Defining your own commands is a powerful tool, but there are problems if you define commands you want to use over a long period of time. Entering the commands every time you launch `gdb` becomes annoying after a while. Plus, you could make a typographical error, forcing you to reenter the command.

The solution is to place the commands you want to define in a file and load the file when you launch `gdb`. Place one `gdb` command per line. The character `#` signifies a comment. `gdb` ignores blank lines and tabs so you can indent commands to make them easier to read. You can also enter documentation for your commands in the file. Make sure to define the command first. The example below shows how the `PrintArray` command would look in a file.

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end

document PrintArray

The PrintArray command displays the contents of dynamic arrays. It takes
three arguments.

$arg0      The array.
$arg1      First element to display.
$arg2      Number of elements to display.

end
```

The `source` command executes commands from a file. Supply a file name.

```
source Filename
```

In the case of user-defined commands, running the `source` command loads the commands and documentation from the file. After loading the commands from the file, you can execute them when needed.

To have your user-defined commands loaded when `gdb` starts, place the commands in a file and perform the following steps:

- 1) Move the file you created to your home directory, `/Users/YourUsername`.
- 2) Run the Terminal program to reach the command line.
- 3) Change the name of your file to `.gdbinit` from the command line using the `mv` command.

The Finder won't let you start a file's name with a period because the operating system allows only system files to begin with a period. To start a file's name with a period, you must change the file's name from the command line.

When `gdb` launches it loads the `.gdbinit` file and executes any commands in the file. In the case of user-defined commands, `gdb` defines the commands on startup so you can start using your commands immediately.



## Command Hooks

A command hook lets you execute a series of `gdb` commands before or after a command executes. You can create command hooks for every `gdb` command as well as user-defined commands. A command hook has the same structure as a user-defined command.

```
define
  Commands in hook
end
```

When defining a command hook to execute before the command, the `define` command takes the following form:

```
define hook-CommandName
```

When defining a command hook to execute after the command, the `define` command takes the following form:

```
define hookpost-CommandName
```

The following example prints a header for the `PrintArray` command example:

```
define hook-PrintArray
  printf "Array Contents\n\n"
end
```

## Chapter 3

# Interface Builder

The name Interface Builder reveals that it's a program to build user interfaces, but Interface Builder can do more than build interfaces. Interface Builder works with Xcode to simplify the development of Cocoa programs. Use Interface Builder to make connections between user interface elements, create subclasses of Cocoa classes, add methods to Cocoa classes, and create source code files.

Only Carbon and Cocoa programs can use Interface Builder. If you're writing a Java program using the AWT or Swing frameworks, you have three options.

- Use nib4j, which is a third party tool to design Swing interfaces using Interface Builder. It's free for non-commercial use.
- Use another Java environment that includes a GUI designer.
- Create user interfaces in your program by writing code.

## Starting with Interface Builder

Normally the way you start with Interface Builder is by creating an Xcode application project and double-clicking the nib file Xcode creates when it makes your project. Double-clicking the nib file opens it in Interface Builder, where you can make changes to your program's user interface.

Remember from Chapter 1 about the two types of Cocoa applications you can make in Xcode: applications and document-based applications. Applications have one window open, and document-based applications can have multiple windows open at one time. Applications initially have one nib file, `MainMenu.nib`. It contains your application's menu bar and window. Document-based applications initially have two nib files. `MainMenu.nib` holds the menu bar, and `MyDocument.nib` contains the window that will be created when your application opens a new document window.

The `MainMenu.nib` file Xcode creates for a Cocoa application contains four items.

- File's Owner.
- First Responder.
- MainMenu, which contains your program's menu bar and menus.
- Window.

Refer to the section "The Nib File Window for Cocoa Applications" to learn about the File's Owner and First Responder items. These items require too much explanation to cover here.

A document-based Cocoa application does not have a Window item in the `MainMenu.nib` file. The Window item appears in the `MyDocument.nib` file.

Carbon applications initially have one nib file: `main.nib`. The `main.nib` file has two items: MainMenu and MainWindow. The File's Owner and First Responder items work with Cocoa classes, which is why they're not part of a Carbon application's nib file.

## Creating User Interfaces for Cocoa Programs

Most of you are using Cocoa so I'm going to start by covering the use of Interface Builder for Cocoa programs. If you're writing a Carbon program, skip ahead to the section "Creating User Interfaces for Carbon Programs".

Before you start building your program's user interface, read Apple's Human Interface Guidelines. Mac users have expectations on how an application should behave, and they expect your application to behave similarly. You can read the guidelines in Xcode's documentation window by selecting the Human Interface Guide bookmark in the window.

### Laying out the Interface

The blank window provided with the nib file Xcode creates is a starting point, not a finished product. To add user interface elements to the window, use the palettes window. The palettes window should be open, but if it's not, choose Tools > Palettes > Show Palettes to open it. You may have to resize the window to see all the toolbar buttons.

There are a lot of user interface elements for Cocoa applications so Interface Builder places them in palettes. Interface Builder puts a button for each palette in the palettes window toolbar. The following palettes make up the basic palette list:

- Address Book palette
- AppleScript palette
- Cocoa menus palette
- Cocoa controls and indicators palette
- Cocoa text controls palette
- Cocoa windows palette
- Cocoa data views palette
- Cocoa container views palette
- Cocoa graphics views palette
- Controllers palette

Interface Builder 2.5, the version that ships with Mac OS X 10.4, adds several new palettes.

- Automator palette
- Disc recording palette
- OSA palette
- PDF Kit palette
- Quartz Composer palette

Placing user-interface items is simple. Select a palette by clicking its toolbar button, and select an item from the palette. For items you want to add to the window, drag the item to where you want it to appear in the window. Some of the items, such as additional windows, cannot be displayed inside a window. You will be unable to drag these items to a window. Drag them to the nib file window.

To create multiple copies of a control, select the control and option-drag. Option-dragging creates a matrix of controls in the window. Select the matrix and control-drag to change the spacing between the controls in the matrix. If a control cannot be part of a matrix, option-dragging creates a copy of the control instead of a matrix of that control.

### **Adding Palettes to the Palettes Window**

The first time you launch Interface Builder, you won't find the new palettes I mentioned in the previous section. Interface Builder does not add those palettes for you automatically. You must add the new palettes to the palettes window. To add a palette to the Cocoa palettes window:

- 1) Choose Interface Builder > Preferences to open Interface Builder's preferences window.
- 2) Click the Palettes tab.
- 3) Click the Add button.
- 4) Select the palette you want to add and click the Open button. The new Interface Builder 2.5 palettes are in the directory `/Developer/Extras/Palettes`.

Interface Builder does not limit you to the built-in palettes for Cocoa user interfaces. Developers can create palettes of objects other programmers can use in Interface Builder. All you have to do to use these palettes is add them to the palettes window.

### **Address Book Palette**

The Address Book palette has only one control. It lets your application work with the Mac OS X Address Book application. Mac OS X's Mail application demonstrates the use of the Address Book palette. Choosing Window > Address Panel opens the address panel. Selecting a name from the address panel and clicking the To button addresses an email to that person.

### **AppleScript Palette**

The AppleScript palette contains one item, an ASKDataSource item (ASK stands for AppleScriptKit). The data source object supplies data to tables. You cannot drag this item to a window. Drag it to the nib file's window.

### **Cocoa Menus Palette**

The Cocoa menus palette lets you add menus to your application's menu bar and add items to your menus. Interface Builder places five menus in the menu bar for a new Cocoa application.

- Application menu
- File menu
- Edit menu
- Window menu
- Help menu

In addition to the application, File, Edit, and Window menus, you can add the following menu-related items:

- Find menu
- Font menu
- Text menu
- Format menu
- Submenus
- Menu items
- Menu separators
- NSMenu object

Use one of the named menus to add a menu to the menu bar, even if your application needs a menu that's not in the list. Change the menu title to the title you want.

Submenus work with hierarchical menus. Choose File > Open Recent to see an example of a hierarchical menu. The submenu contains the most recent files you opened in Interface Builder. Menu separators group related items in a menu. If you have a lot of items in a menu, use menu separators to make finding an item in the menu easier.

Dock menus and contextual menus use the `NSMenu` object. A Dock menu appears when you click a Dock icon and hold the mouse button. For each running application, the Dock menu has items to quit and hide the application as well as items for each open window in the application. Use the `NSMenu` object to create a custom Dock menu for your program. Contextual menus work with user interface elements. The menu opens when the user control-clicks the element. The contextual menu contains common commands for the user interface element.

## Cocoa Controls and Indicators Palette

The Cocoa controls and indicators palette contains all the controls that don't involve the user entering text. The following controls reside in the Cocoa controls and indicators palette:

- Buttons. There are many kinds of buttons.
- Radio buttons.
- Checkboxes.
- Pop-up buttons, which contain pop-up menus.
- Sliders.
- Progress indicators.
- Steppers.
- Color wells, which show the current color and let the user choose a color.
- Segmented controls.
- Level indicators.

A segmented control is a control that is divided into multiple segments. The back and forward buttons at the top of Finder windows demonstrate the use of segmented controls. The buttons are the two segments in the segmented control. Apple added the segmented control to Interface Builder 2.5, the version that ships with Mac OS X 10.4. Segmented controls work on Mac OS X 10.2 and 10.3.

Level indicators indicate amounts graphically. A common use of level indicators is measuring the relevance of search results. Apple introduced level indicators in Mac OS X 10.4; earlier versions of Mac OS X cannot use them.

## Cocoa Text Controls Palette

The Cocoa text controls palette contains the controls that let the user type text in your program and the controls that display text. The palette contains the following controls:

- Text fields, where the user enters text.
- Search field, where the user enters something to search for.
- Forms, which are groups of text fields.
- Text views, where the user enters large amounts of text. A word processor would use a text view for the user to type documents.
- Combo boxes, which combines a text field with a list.
- Static text fields. Interface Builder supplies four different text sizes for static text fields.
- Date formatters.

- Number formatters.
- Date pickers.
- Token fields.

Use forms instead of individual text fields when you have groups of related text items. Suppose your program requires the user to enter contact information: name, address, phone number, and email address. Rather than create a text field for each piece of contact information, create one form for all the contact information. Forms save you work when working with multiple text fields.

The date formatter controls how your program displays dates, and the number formatter controls how your program displays numbers. Drag the formatters to tables and the controls in the Cocoa text controls palette.

A date picker is a control to choose the date and time. There are two date picker styles. The textual style uses a text field and steppers. The graphical style uses a calendar and clock. Apple introduced the date picker in Interface Builder 2.5, the version that ships with Mac OS X 10.4. Previous versions of Mac OS X cannot use date pickers.

A token field creates a token out of text. When you enter the email address of someone in your address book in Mac OS X's Mail application, Mail creates a token with the person's name. Apple introduced token fields in Mac OS X 10.4.

## **Cocoa Windows Palette**

The Cocoa windows palette contains the windows you can add to a Cocoa program. The palette has four items.

- Window, which you can use for dialog boxes and sheets as well as windows.
- Panel.
- Drawer, which is a view you attach to a window. The user controls whether the drawer is open or closed. Sampler uses a drawer to set the sampling rate and specify the functions to watch.
- Window with a drawer attached to it. This item adds a window, a drawer, and a view for the drawer to the nib file.

To add windows to your program's user interface, drag them to the nib file's window.

Looking at the windows palette, you can see windows and panels are both windows so what's the difference between them? Panels are auxiliary windows, windows you don't need to use often. To see an example of a panel, open the font panel by choosing Format > Font > Show Fonts. Working with fonts isn't something you're going to do often when building user interfaces, which makes fonts perfect for a panel.

If you're going to use drawers in your program, the easiest way to do so is to use the window with the drawer attached. The window with drawer attached takes care of most of the details of dealing with drawers. All you need to do is add a button or a menu item to open and close the drawer. To add a drawer to an existing window:

- 1) Drag a drawer object to the nib file's window.
- 2) Drag a custom view object from the container views palette to the nib file's window.
- 3) Make a connection from the drawer to the window by control-dragging from the drawer to the window. This connection will make the drawer's parent window the window you're attaching the drawer to.
- 4) Make another connection from the drawer to the custom view to make the drawer's content view the custom view.

For more information on making connections, refer to the section "Making Connections" later in the chapter.

## Cocoa Data Views Palette

The Cocoa data views palette contains tables and other views to display large amounts of data in your program. The palette contains the following items:

- Table views
- Outline views
- Browsers
- Text cells for tables
- Button cells for tables
- Slider cells for tables
- Stepper cells for tables
- Image cells for tables
- Pop-up button cells for tables
- Combo box cells for tables

A table view contains rows and columns of data. The detail view of Xcode's documentation window is an example of a table view. An outline view is a table view that has disclosure triangles to expand and collapse rows of the table. The variable viewer in Xcode's debugger window is an example of an outline view. Browsers display lists of data. If the data is not hierarchical, the browser has one column. If the data is hierarchical, the browser can display one column for each level of the hierarchy.

To see the difference between browsers and the table and outline views, go into the Finder. Choosing View > As List tells the Finder to use an outline view in Finder windows. Choosing View > As Columns tells the Finder to use a browser in Finder windows. When you view as list each folder and file has four columns of information: Name, Date Modified, Size, and Kind. You can sort the folders and files by any of these columns. When you view as column, there are no Date Modified, Size, and Kind columns. The files and folders are sorted alphabetically. Selecting a folder shows the folder's contents in the next column.

To use one of the table cells, you must first drag a table view or an outline view to the window. After you create the table, drag the cell to the column where you want the cell to appear in the table.

## Cocoa Container Views Palette

The Cocoa container views palette contains views that hold other user interface elements. Instead of dragging the elements to the window itself, drag a container view to the window. Drag the user interface elements to the container view. There are four container views.

- Custom views
- Group boxes
- Tab views
- Image wells

Interface Builder provides views for OpenGL and QuickTime as well as a view for displaying web pages. Those views are in the Cocoa graphics views and QTKit palettes. If you have special drawing needs, there's the custom view. The custom view item is a placeholder for a view subclass that you create. Place the custom view in the window. Create a subclass of Cocoa's view class, `NSView`, and tell Interface Builder the custom view is the subclass you created.

Group boxes divide a window into distinct areas. If your window has lots of user interface elements, using group boxes makes your interface easier to comprehend. If you're going to use a group box, start by creating the group box. Drag the controls into the group box.

Tab views allow you to show multiple pages of information in a single window. Selecting a tab makes that tab's page appear in the window. Interface Builder's nib window is an example of a tab view. It has five tabs for Cocoa programs: Instances, Classes, Images, Sounds, and Nib.

The initial tab view has two tabs when you place it in a window. To add more tabs to the tab view:

- 1) Select the tab view.
- 2) Open the tab view's information panel by choosing Tools > Show Info.
- 3) Select Attributes from the pop-up menu at the top of the information panel.
- 4) Enter the number of tabs you want in the Number of Items text field.

To place controls in a particular tab, select the tab. Drag the controls you want to the tab's view. Repeat for each tab.

Image views display an image in a frame. You can also use an image view as a target for the user to drag an image.

### **Cocoa Graphics Views Palette**

Graphics views let your Cocoa programs work with Apple's graphics technologies. They provide a destination for your application's drawing. Instead of drawing into the window, you attach a graphics view to the window by dragging the view from the palette to the window in Interface Builder. Your program draws into the view you attached to the window. Interface Builder includes two views in the graphics views palette.

- Web views.
- OpenGL views, which OpenGL Cocoa programs should use.

Web views provide a destination for displaying Web pages. If your application shows dynamic content from the Internet, you will be using Web views.

### **Controllers Palette**

Cocoa uses the Model-View-Controller (MVC) design pattern. Model classes store data. View classes are the visible user interface items. Controller classes communicate between model and view classes. They send data from model classes to view classes and vice versa. Controller classes are invisible to the user.

The controllers palette lets you add controller objects to your Cocoa program. Controller objects take care of most of the work involved in communicating between model and view classes. Apple added controller objects in Mac OS X 10.3, which means you cannot use controller objects with earlier versions of Mac OS X. The controllers palette has five controllers.

- Object controllers manage one object.
- Array controllers manage multiple objects.
- The user defaults controller works with applications that save user preferences. The user defaults database stores the preferences that differ from the ones that shipped with the application.
- Managed object contexts manage a collection of managed objects. They work with the Core Data framework.
- Tree controllers manage a tree of objects.

Apple added managed object contexts and tree controllers in Mac OS X 10.4. Earlier versions of Mac OS X cannot use them.



Because controllers aren't visible, you can't drag controller objects to windows. Drag them to the nib file's window. If you're going to add multiple controllers, you should rename the controllers so you can keep track of them.

## **Automator Palette**

The Automator palette has elements that work with Mac OS X 10.4's Automator technology, which lets you perform tasks in other applications automatically. The palette contains three pop-up buttons for choosing applications, directories, and files.

## **Disc Recording Palette**

The disc recording palette has one control, a Minutes/Seconds/Frames formatter (MSF formatter). The MSF formatter calculates the length of CD and DVD tracks. CDs and DVDs store their data in frames, where 75 frames equal one second. The MSF formatter displays the frames in the human-readable form of minutes and seconds.

You cannot drag the MSF formatter to the window; the MSF formatter works with controls that display text to format the text. Controls that work with formatters include text boxes, forms, search fields, combo boxes, and tables. First, place a control that works with formatters, such as a text box, in the window. Then drag the MSF formatter to the control. For tables, drag the formatter to the column heading. If a control cannot handle a formatter, Interface Builder will reject your attempt to drag a formatter to the control.

If you have multiple controls that want to use an MSF formatter, drag the formatter to the nib file window. Multiple controls can then access the formatter.

Some versions of Interface Builder call this palette the Minutes/Seconds/Frames (MSF) formatter palette.

## **OSA Palette**

The Open Scripting Architecture (OSA) palette works with AppleScript. This palette has four items.

- Scroll views for viewing and editing scripts.
- Dictionary views for viewing the AppleScript terms an application supports.
- Script controllers for managing scripts.
- Dictionary controllers for managing dictionaries.

## **PDF Kit Palette**

The PDF Kit palette contains one item, a PDF view for displaying and changing PDF documents.

## **QTKit Palette**

The QTKit palette contains one item, a movie view to display QuickTime movies in your Cocoa programs. Some versions of Interface Builder place the movie view in the Cocoa graphics views palette.

## Quartz Composer Palette

The Quartz Composer palette has elements that work with the Quartz Composer application. The palette contains two items: a Quartz Composer view and a Quartz Composer patch controller. Use the Quartz Composer view to play a Quartz Composer composition in a Cocoa program. Use the Quartz Composer patch controller to create bindings between compositions and user interface elements.

## Customizing the Palette Window Toolbar

When you add a palette to the palette window, it appears at the left end of the palette window toolbar, which may not be where you want it to be. To rearrange the items in the palette window toolbar:

- 1) Control-click one of the toolbar buttons.
- 2) Choose Customize Toolbar from the contextual menu to open the customize toolbar window.
- 3) Select a button from the toolbar and drag it to where you want it to appear in the toolbar.
- 4) Click the Done button when you're finished rearranging the toolbar buttons.

## Modifying the Interface

When you lay out your program's user interface, the interface isn't going to look right initially. Buttons, checkboxes, and radio buttons display placeholder text. The default size for views is pretty small. You'll want to make them bigger. This section shows you how to make the modifications your user interface needs.

You can make basic changes to your interface easily. Click on a control or a view inside a window to select it. Figure 3.1 shows what a selected item looks like. Dragging the selected item changes its position in the window. To resize a control or view, click one of the eight dots bordering the control you selected, hold down the mouse button, and move the mouse.

To change the text in a control, double-click the text and enter the new text. To make other changes you must use the information panel. Choose Tools > Show Info to open the information panel. Most items in a Cocoa nib file have the following items in the pop-up menu at the top of the panel:

- Attributes
- Connections
- Size
- Bindings
- Custom Class
- Accessibility
- Help
- AppleScript
- Sherlock



**Figure 3.1**

Selected item.

## Attributes Panel

The attributes panel is where you set an item's basic attributes, anything not covered by the other panels. Use the attributes panel to perform the following tasks:

- Give menu items and controls keyboard equivalents.
- Provide titles for windows and controls.
- Set the number of rows and columns in a radio button group.
- Determine the radio button that should be initially selected in a radio button group.
- Determine whether or not a checkbox should be selected initially.
- Set a slider's minimum, maximum, and initial values.
- Set the number of tabs in a tab view.
- Set the number of columns in a table.
- Give a text field placeholder text to display initially.
- Determine the controls a window should have: close button, minimize button, zoom button, and resize control.

Each item in the Interface Builder palettes has its own set of attributes, which means what you can set in the attributes panel depends on the item.

## Connections Panel

Cocoa programs use messages to send data from one object to another. Connections let your user interface elements send messages to each other, send messages to model and controller objects, and receive messages from model and controller objects. The connections panel, shown in Figure 3.2, is where you manage your user interface elements' connections. Looking at the figure, you can see two tabs: Outlets and Target/Action.



**Figure 3.2**

Connections panel.

The Outlets tab shows you the possible outlets this particular item can have. Outlets are variables in a class that can connect to other classes or controls. If an outlet has a connection, there will be a gray dot next to it, and the destination column will contain the item this outlet is connected to.

Initially the Target/Action tab will be blank because you haven't added any connections. When you add a connection, the tab will fill up with the possible actions this particular connection can have. Figure 3.3 shows an example. When two items make a connection, there can be only one action. That action will have a gray dot next to it. To change the action, select the action you want and click the Change Action button at the bottom of the window. If you haven't made an action selection, the button will say Connect. Refer to the section "Making Connections" for more information on connections.

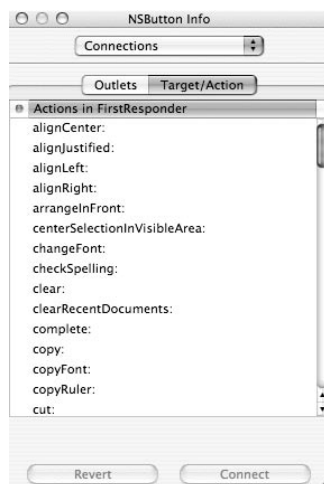
## Size Panel

If you guessed the size panel lets you modify an object's size, you're correct. The size panel (look at Figure 3.4 for an example) lets you specify an object's size and position. Most of you will find it easier to select the object in the window and use the mouse to move or resize the object. The size panel is good for making minor modifications, such as making an object one pixel bigger or smaller.

Looking at Figure 3.4, you can see two tabs titled Frame and Layout. The Frame tab shows the object's size and position including its frame. The Layout tab shows the size and position without the frame. If an object doesn't have a frame around it, the size and position will be identical with both tabs.

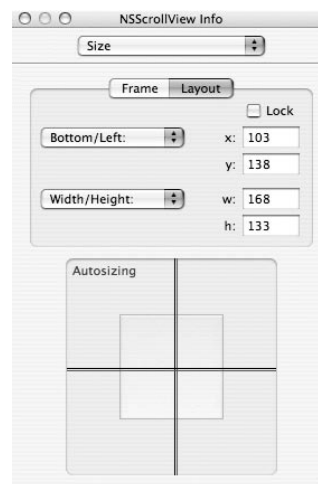
## Autosizing

Most items have an autosizing area at the bottom of the size pane, which you can see if you look at Figure 3.4. The autosizing area consists of two squares, one inside the other, with a cross in the center of each square. Clicking a piece of a cross will make it springy (curly). By fiddling around with the pieces of the cross, you determine what the item will do when the user resizes the window.



**Figure 3.3**

Target/Action tab.



**Figure 3.4**

Size panel.

If all the pieces are rigid (straight), which is the default, the object maintains its size and position relative to the window when the window resizes. If you have a control at the bottom of the window, it will remain at the bottom of the window. Springy pieces in the inner square change their size when the window resizes. Objects can resize vertically, horizontally, or in both directions. Views are the most common items you would want to resize along with the window. In an Xcode source code window, you want the area where you type code to get bigger when you make the window bigger.

Springy pieces in the outer square mean the object keeps its literal position when the window resizes, which makes the relative position of the object change. Let's say you have a control at the bottom of a window, and you make the bottom portion of the outer square springy. When the user makes the window shorter, the control disappears. The amount of space between the control and the bottom of the window increases when the user makes the window taller.

Objects can change their relative directions in four directions: up, down, left and right. Objects close to an edge of the window may not change in certain directions. If you have a control near the bottom of the window and only the up direction set to move (springy), the object will maintain its relative position. Objects close to the left edge won't move right much. Objects close to the right edge won't move left much, and object close to the top of the window won't move down much.

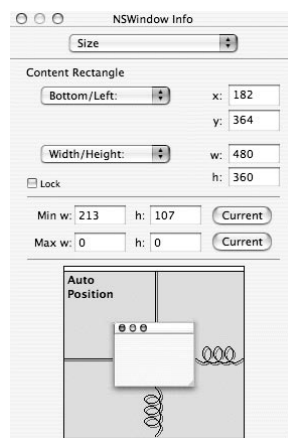
## Size Panel for Windows

A special case of the size panel is for windows, which you can see in Figure 3.5. In the center of the panel, you can set the minimum and maximum size the user can make the window. Clicking one of the Current buttons makes the current window size the maximum or minimum depending on which button you click. You should set a minimum size and make it large enough to fit the window's controls.

## Auto Positioning Windows

The Auto Position area determines where the window appears when the user opens the window. There are four pieces, one piece for each edge of the window. Each piece can be either springy or rigid. Click on a piece to change it from rigid to springy and vice versa.

If a piece is rigid, the window remains a fixed distance from the edge of the screen. The distance from the screen's edge depends on where you position the window in Interface Builder. If a piece is springy, the window repositions itself to keep a proportional distance from the edge of the screen. The proportional distance depends on your monitor's screen resolution and where you position the window in Interface Builder.



**Figure 3.5**

A window's size panel.

Suppose you position a window 128 pixels from the left edge of the screen and your screen width is 1280 pixels. If the left edge is rigid, the window will always appear 128 pixels from the left edge of the screen when the user opens the window. If the left edge is springy, the window's initial position depends on the screen width of the user's monitor. The position will be ten percent of the screen width in this example because 128 is ten percent of 1280. If the user has a screen width of 1600 pixels, the initial position will be 160 pixels from the left edge of the screen.

Interface Builder initially auto positions windows so the top and left edges of the window are a fixed distance (rigid pieces) from the top and left edges of the screen. The bottom and right edges of the window are a proportional distance (springy pieces) from the bottom and right edges of the screen. Interface Builder's positioning works well in most situations. You would use a proportional distance for all four edges if you wanted the window centered on the screen.

## Bindings Panel

In the Model-View-Controller design pattern, programmers spend a lot of time writing code for controller classes that supply data from model classes to view classes and vice versa. Cocoa bindings reduce the amount of code you have to write. Cocoa has controller classes, which you can find in the controllers palette, that save you from writing code. Bindings connect user interface elements to these controller classes.

If you're going to use bindings in your program, keep in mind that Apple introduced bindings in Mac OS X 10.3. Earlier versions of Mac OS X cannot use bindings.

The bindings panel is where you connect the items in your user interface to controller objects. Refer to the section "Creating Bindings" to learn how to make a binding.

## Custom Class Panel

Every item in the Interface Builder palettes has a corresponding Cocoa class. The custom class panel shows the item's Cocoa class and for some classes it shows related classes, such as classes it inherits from and classes that inherit from it. Double-clicking a class opens the class's information panel, shown in Figure 3.6. If you read the section "Connections Panel" earlier in the chapter, you may remember that each item has its own outlets and actions. The information panel lets you examine the class's outlets and actions.



**Figure 3.6**

Cocoa class information panel.

## Accessibility Panel

The accessibility panel lets you set attributes to make your program accessible to people with disabilities. The AXDescription text field describes the element. Suppose your program has back and forward buttons that display arrows instead of text. By giving the buttons descriptions, a blind person would be able to understand what your buttons do.

The AXHelp text field sets the accessibility equivalent of a tool tip. Enter the help text in the text field.

The bottom of the accessibility panel is similar to the connections panel. Each Interface Builder element has an accessibility object associated with it. The bottom of the panel contains the accessibility object's connections to other accessibility objects. Instead of outlets and actions, accessibility objects have attributes. To connect two elements for accessibility:

- 1) Hold down the Control key and click the item you want to initiate the connection.
- 2) Drag the mouse to the item you want to receive the connection.
- 3) Select the attribute you want to set in the accessibility panel.
- 4) Click the Connect button.

## Help Panel

The help panel is where you enter the text for a control's tool tip. When the user leaves the mouse cursor over the control, the tool tip appears, providing additional help.

## AppleScript Panel

The AppleScript panel is where you connect AppleScript event handlers to user interface elements. Figure 3.7 shows a sample AppleScript panel. The contents of the panel depend on the user interface element; each element has its own list of events it can handle.

The AppleScript panel organizes the event handlers into groups. Selecting the group's checkbox turns on every event handler for that group. Clicking the Mouse group turns on the event handlers for all mouse events. If you're interested in handling only some of a group's events, click the disclosure triangle to show all the events and select the events you want to handle.



**Figure 3.7**

AppleScript panel.

Cocoa has default event-handling scripts for all the events you can set in the AppleScript panel. You can use these scripts even if you write all your Cocoa code in Objective C or Java. The built-in event handlers work fine in general cases, but your program may need to handle some events in a special way. You must write your own scripts for the events that require special handling.

If you have the nib file's project open in Xcode and the project has AppleScript files in it, the bottom half of the panel will show a list of your project's script files. Select the checkbox next to the file where you want the event-handling script to appear. Select the event you want to write an event handler for and click the Edit Script button. The skeleton for the event handler appears in the AppleScript file in Xcode. You're responsible for writing the event handler's code. Selecting a group of events and clicking the Edit Script button creates event handlers for each event in the group.

You can also create new AppleScript files from Interface Builder. Click the New Script button and a dialog box opens asking you where you want to create the file. If you want to add the file to your project, make sure the project is open in Xcode. If the project isn't open, you can create the file, but you won't be able to add it to the project from Interface Builder.

### **Sherlock Panel**

Sherlock channel projects use the Sherlock panel. The Sherlock panel is where you give each user interface element a name Sherlock can use and a data store path. The data store path is a unique label for an object in a Sherlock channel.

The user interface for a Sherlock channel has a top-level view on which you place all the other user interface elements. Give the top-level view the same name and data store path. Xcode Sherlock channel projects give the top-level view the name and data store path Internet.

When you name an element in the Sherlock pane, Interface Builder adds the name to the data store path as well. Interface Builder places the top-level view's name in front of the element to give the element's data store path a unique name. The element's data store path takes the following form:

`TopLevelViewName.ElementName`

### **Creating Source Code in Interface Builder**

One of the cool things about using Interface Builder for Cocoa development is being able to generate source code in Xcode from Interface Builder. Among the tasks you can perform in Interface Builder include:

- Creating subclasses of Cocoa classes.
- Adding actions to Cocoa classes.
- Adding outlets and actions to the subclasses you create.
- Adding instances of subclasses to the nib file.
- Creating files in Xcode for the subclasses you create.



## Creating Subclasses

To create a subclass of a Cocoa class, select the **Classes** tab in the nib file window, which will make the window look like Figure 3.8. In this view you can navigate the Cocoa classes to find the one you want to subclass. The classes you will subclass most often are the Cocoa view classes and the `NSObject` class. `NSObject` is the base class for all Cocoa classes. Subclass `NSObject` to create model classes.

After selecting the Cocoa class you want to subclass, choose **Classes > Subclass** to create the subclass. The subclass has the prefix `My`; subclassing `NSObject` gives you a class name `MyObject`, but you can rename it if you want.

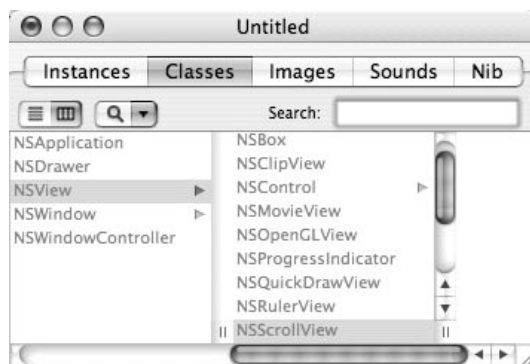
## Adding Outlets and Actions

Any subclass you create inherits the outlets and actions of the class from which it inherits. *Outlets* are variables in a class you can link to other objects in your program. *Actions* are methods (member functions) of an object. Other objects send the object a message to perform an action. Clicking a **Print** button would send a message to an object telling it to print.

To add outlets and actions to a class, select the class in the nib file window. Choose **Tools > Show Info**, and the class's information panel opens, which you can see in Figure 3.9. The panel contains two tabs listing the number of outlets and actions the class has. To add an outlet, click the **Outlets** tab and click the **Add** button. A new outlet titled `myOutlet` appears in the window; change it to the name you want. Adding an action works the same way, except the new action will have the name `myAction`. You can also add outlets and methods by choosing **Classes > Add Outlet** or choosing **Classes > Add Action**.

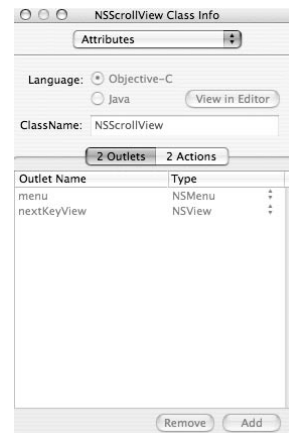
### NOTE

You cannot add outlets to any of the built-in Cocoa classes. Cocoa allows you to add methods to the built-in classes, but not variables, and outlets are variables. You can add outlets only to subclasses you create.



**Figure 3.8**

Viewing classes in the nib file window.



**Figure 3.9**

Cocoa class information panel.

**Creating Instances in the Nib File**

After creating a subclass and adding outlets and actions to it, you want to use the subclass in Interface Builder. Earlier I mentioned that view classes are popular classes to subclass. Interface Builder comes with several views you can drag to a window. Wouldn't it be great if you could use the built-in views, but have them refer to your subclass instead of the Cocoa class?

Miraculously, Interface Builder can refer to the view's subclass. Assuming you've already created the subclass:

- 1) Drag the view you subclassed to the window.
- 2) Select the view in the window.
- 3) Choose Tools > Show Info to open the information panel for the view.
- 4) Choose Custom Class from the pop-up menu in the information panel.
- 5) The subclass you made should be one of the available classes in the window. Select it.

Notice how the caption in the view changes to the name of your subclass. For other subclasses you make, you must instantiate them to use them in Interface Builder.

- 1) Select the Classes tab in the nib file window.
- 2) Select your class in the window.
- 3) Choose Classes > Instantiate.

The nib file window view will change from Classes to Instances, and you will see an instance of your class appear in the window.

**Creating Source Code Files**

After creating your subclasses and adding outlets and actions to the subclasses, all that's left to do is create the files for the subclasses. Make sure you have your project open in Xcode so Interface Builder can add the files to your project.

- 1) Select the Classes tab in the nib file window.
- 2) Select your class in the window.
- 3) Choose Classes > Create Files to open the create files dialog box. Assuming you're using Objective C, Interface Builder will create a header file (.h) and an implementation file (.m) for the class. The lower right portion of the dialog box contains a list of open Xcode projects. If the list is empty, don't panic. You can add the files to your project in Xcode. The dialog box should be pointing to your project's folder, but if not, use the dialog box to navigate to the project folder.
- 4) Click the Choose button to create the files.

To see what Interface Builder created for you, go to Xcode and look at your project window. The files Interface Builder created for you should be there for you to examine. If not, choose Project > Add to Project to add the files to your project. Open the header file. Notice how the outlets you created in Interface Builder appear in the class declaration with type `IBOutlet`. Also notice how the actions you created appear in the methods list, returning type `IBAction`. The implementation file has empty methods for you to fill in.

## Parsing Xcode Header Files into Interface Builder

In the previous section I showed you how to use Interface Builder to generate code in Xcode. You can reverse the process by writing subclasses, actions, and outlets in Xcode, then dragging the header file to the nib file in Interface Builder. Dragging the header file adds the subclass to your nib file, and the information panel for the subclass will display the outlets and actions you wrote. Alternatively, you could choose **Classes > Read Files** in Interface Builder and select the header file instead of dragging the header file from Xcode to Interface Builder.

I recommend initially creating your subclasses, outlets, and actions in Interface Builder and generating the source code files from Interface Builder. If you make a typing error in Xcode, your outlets and actions may not appear in Interface Builder. When would you want to add outlets and actions in Xcode first? Suppose you created a subclass, outlets, and actions in Interface Builder, created the files in Interface Builder, and wrote code in Xcode to implement the actions. At a later date you decide to add more outlets and actions to your class. If you add them in Interface Builder and choose to create files, Interface Builder will overwrite the old versions of your class files, wiping out the code you wrote to implement the original actions. In this situation you would want to add the new outlets and actions in Xcode, then read the updated header file into Interface Builder.

## Making Connections

When you want one object to be able to send a message to another object, make a connection between the two objects. To make the connection:

- 1) Hold down the Control key and click the item you want to send the message. As long as you have the mouse button down, you can let go of the Control key.
- 2) Drag the mouse to the item you want to receive the message. While dragging, a line will appear that signifies the connection.

After establishing the connection with the control-drag, the information panel for the item initiating the connection opens. The panel shows the possible actions or outlets for the item on the receiving end of the connection.

- 3) Select the action or outlet you want for the connection.
- 4) Click the Connect button at the bottom of the information panel.

The action or outlet you selected will have a gray dot next to it to show it's the action to perform for this connection.

## Creating Bindings

Bindings allow your Cocoa program to use controller objects to manage relationships between model and view objects. They reduce the amount of code you have to write. Creating a binding requires you to perform the following tasks:

- 1) Create the model class.
- 2) Create the controller.
- 3) Bind the model to the controller.
- 4) Bind the view to the controller.

### **Creating the Model Class**

A binding requires three objects: a model, a view, and a controller. The Interface Builder palettes contain the view and controller objects while you create the models. To create a model class in Interface Builder:

- 1) Subclass `NSObject` to create the model class. Refer to the section “Creating Subclasses” for detailed instructions on creating subclasses.
- 2) Create a source code file for the model class. Refer to the section “Creating Source Code Files” for detailed instructions on creating source code files.
- 3) Add instance variables and accessor functions to the model class.
- 4) Create an instance of the model class in Interface Builder. Refer to the section “Creating Instances in the Nib File” for detailed instructions on creating instances.

You must use Xcode to add the model class’s instance variables and accessor functions. The instance variables are what the controller binds to the Interface Builder views.

### **Creating the Controller**

Creating the controller is the easiest task to perform. Select a controller from the controllers palette and drag it to the nib file window.

### **Binding the Model to the Controller**

When you create a binding in Interface Builder, the binding needs to know about the model object. Binding the model to the controller provides the necessary information about the model. To bind the model to the controller:

- 1) Create a connection from the controller to the model class. Refer to the section “Making Connections” for detailed instructions on creating connections.
- 2) Add keys for the model class to the controller.

The keys are instance variables in the model classes the controller can connect to an Interface Builder item. Select the controller from the nib file window. Open the information panel and select Attributes from the pop-up menu. Enter the name of the model class in the Object Class Name text field. Click the Add button to add a key to the list. Double-click the name and change it to one of your model class’s instance variables. Add the remaining keys.

## Binding the View to the Controller

After binding the model to the controller, the last task is binding the view to the controller using Interface Builder.

- 1) Open the view object's information panel.
- 2) Choose Bindings from the information panel's pop-up menu. Refer to Figure 3.10.
- 3) Select a binding. Each user interface element has its own set of bindings.
- 4) Choose the controller from the Bind to pop-up menu.
- 5) Choose the Controller Key, which is the key in the controller class. Object controllers and array controllers have their own set of keys. The keys should appear in the combo box.
- 6) Choose the Model Key Path, which is the key in the model class. The keys you created to bind the model to the controller should appear in the combo box.

After binding the view to the controller, you can add a value transformer and placeholders to the binding. Select the Bind checkbox to activate the controls to set the value transformer and placeholders.

## Value Transformers

*Value transformers* are functions that change the value of a piece of data. If a binding uses a value transformer, the model's data goes through the value transformer before the user interface element displays the data. The Cocoa framework comes with four value transformers.

- `NSNegateBooleanTransformerName`, which you should use to select checkboxes and radio buttons.
- `NSIsNilTransformerName`, which you can use to enable and disable user interface elements.
- `NSIsNotNilTransformerName`, which you can use to enable and disable user interface elements.
- `NSUnarchiveFromDataTransformerName`, which you should use with a user defaults controller.

Subclass `NSValueTransformer` to create custom value transformers that perform data conversions. Suppose you store a piece of data in your program as a hexadecimal number, but want to display it as a decimal number. The custom value transformer makes the hexadecimal to decimal conversion.

The bindings panel has a combo box to select a value transformer. The combo box's menu contains the available built-in value transformers for the view you're binding. If you want to use a custom value transformer, you must type the transformer's name in the combo box.

Selecting the Raises for not applicable keys checkbox tells Interface Builder to raise an exception in your program if the controller key or model key path does not apply for a piece of data.



**Figure 3.10**

Bindings panel.

## Placeholders

*Placeholders* display placeholder values in user interface elements when there's no value to display from the model data. Cocoa has default placeholder text, but you can customize the placeholder text your program shows. There are four placeholders.

- Multiple values placeholder, which is the text that appears if the user selects multiple items.
- No selection placeholder, which is the text that appears if the user has not made a selection.
- Not applicable placeholder, which is the text that appears if the controller does not support selection.
- Null placeholder, which is the text that appears if the data is null.

Use the text fields in the information panel to enter the placeholder text. Some bindings do not allow you to customize the placeholder text. If a binding does not allow customized placeholder text, there will be pop-up menus instead of text fields. Use the pop-up menus to determine whether or not placeholder text should appear.

## Working with Menus

Every Mac OS X program with a GUI uses menus. Menus work differently than other user interface elements. You don't drag menu items to a window or to the nib file window. Because menus are so prevalent in Mac OS X programs and require special handling, I've placed the menu building material in its own section.

### Adding Menus to the Menu Bar

To add menus to your application's menu bar, the menu bar must be open in Interface Builder. Double-click the MainMenu item in the nib file window to open the menu bar. Select a menu from the Cocoa menus palette and drag it to the menu bar to add the menu. If the menu isn't in the right location, select it from the menu bar and drag the menu to the proper location.

Looking at the Interface Builder menu bar, you can see several menus — Classes, Layout, and Tools — that aren't in the Cocoa menus palette. How do you add menus that aren't in the palette to the menu bar? Drag one of the menus from the palette to the menu bar, and double-click the menu bar to change the menu name to what you want. From there you can add, remove, and modify the menu items in your new menu to reflect your application's needs.

### Adding Menu Items

Adding items to a menu is simple. Choose the entry in the palette with the name Item (or the blank entry to add a menu separator). Drag the item to the menu in the menu bar where you want to place the menu item. Dragging the item to the menu opens the menu and shows you the items in the menu. Drag the item to where you want it to appear in the menu. If you're adding an item to a pop-up menu, double-click the menu to open the menu before adding the item.

The menu item you added has the name Item, which is most likely not the name you want. Double-click the item to change the menu item's text. You can also change the names of existing menu items. Remember from the previous section that you can drag a menu from the palette and change its name to make it your own menu. If you change the menu's name, you'll want to change the menu items as well.

A special menu item is a submenu, a menu item that contains additional menu items. You can see an example of a submenu in Interface Builder's File menu. The menu item Open Recent is a submenu with the additional items being the nib files you opened most recently. Adding a submenu is the same as adding an ordinary menu item. Drag the submenu to the menu. After you add the submenu to the menu, you can add items to the submenu.

Many standard Mac menu items have keyboard equivalents. Pressing Command-Q is the equivalent of choosing Application > Quit. To give a menu item a keyboard equivalent:

- 1) Select the menu item.
- 2) Choose Tools > Show Info to open the menu item's information panel.
- 3) Choose Attributes from the pop-up menu at the top of the information panel.
- 4) Enter the keyboard equivalent in the Key Equivalent text field. There's a pop-up menu for special keys like the function keys and the Esc key.
- 5) If your keyboard equivalent uses modifier keys, select the appropriate Key Modifier checkboxes. There are four checkboxes running from left to right: Command, Shift, Option, and Control.

To delete a menu item, select it and choose Edit > Delete (or press the Delete key). Deleting menus from the menu bar works the same way. Select the menu from the menu bar and choose Edit > Delete to remove the menu from the menu bar along with all the menu's items.

### Creating a Dock Menu

Every application in the Dock has a menu the user can display by control-clicking the application's icon. If the application is running, the Dock menu has items to quit and hide the application as well as items for each open window in the application. These items appear automatically for your program as well. Creating a Dock menu with custom menu items requires you to take the following steps:

- 1) Drag a `NSMenu` object to the nib file window.
- 2) Add the items you want to add to the menu. Add only the custom items that aren't part of the normal Dock menu.
- 3) Make a connection from the File's Owner to the `NSMenu` object. Hold down the Control key, select the File's Owner in the nib file window, and drag the mouse to the menu.
- 4) When you make the connection, the connections panel for the File's Owner opens. Select `dockMenu` and click the Connect button.
- 5) Add the key `AppleDockMenu` to your program's `info.plist` file. The value for this key is the name of the nib file. Skip the `.nib` extension when entering the nib file name.

### Creating a Contextual Menu

Contextual menus work with user interface elements. When the user control-clicks the element, the contextual menu opens. The contextual menu contains frequently used commands. If you're writing a text editor and the user selects some text, a contextual menu for the text view would have items for cutting, copying, and pasting text. The point of contextual menus is to keep the user from having to move the mouse cursor to the menu bar to perform a menu command. To create a contextual menu:

- 1) Drag a `NSMenu` object to the nib file window.
- 2) Add the items you want to add to the menu.
- 3) Make a connection from the control to the `NSMenu` object. Hold down the Control key, select the control, and drag the mouse to the menu.
- 4) When you make the connection, the connections panel for the control opens. Select `menu` and click the Connect button.

## Testing Your Interface

After you have your interface set, you'll want to give it a test drive to make sure everything looks right. You can test your interface in Interface Builder by choosing File > Test Interface. Interface Builder will simulate your interface as though it were a Mac OS X application. If you have text boxes in your interface, you can type in information, cut and paste, and use the Tab key to move to the next text box. You can use the Page Setup menu to set up for printing. You can save your window to a PDF file, print it, and even fax a copy of it if you want. If you created any tool tips for a control to provide help, your tool tip text will appear if you keep the mouse cursor over that control.

Stopping the interface test and returning to Interface Builder can be confusing. The cause of the confusion is that during interface testing, the menu bar changes to your interface's menu bar, but the menu bar has the title Interface Builder for the application menu instead of your application's name. Don't let the title confuse you; the menus are your interface's, not Interface Builder's. Choosing Quit from the Application menu will stop the interface testing and take you back to Interface Builder.

There is an alternative way of stopping interfacing testing. When testing your interface, the Interface Builder icon changes in the Dock, and a matching icon appears in the menu bar, which you can see in Figure 3.11. Clicking the icon in the menu bar stops the interface testing.

## The Nib File Window for Cocoa Applications

Up to this point in the chapter, I've been covering the objects that make up a nib file: windows, menus, controls, and controllers. The nib file window is where you look at the nib file as a whole instead of looking at its individual pieces. The nib file window for Cocoa programs has five tabs.

- Instances tab
- Classes tab
- Images tab
- Sounds tab
- Nib tab

### Instances Tab

The Instances tab shows the objects in the nib file. A regular Cocoa application has four items in the Instances area: File's Owner, First Responder, MainMenu, and Window. When you add windows, dock menus, contextual menus, and controllers, these objects appear in the Instances area. Creating instances of your subclasses also adds entries to the Instances area.



**Figure 3.11**

The menu bar when testing a Cocoa application's interface.



## File's Owner

The File's Owner is the object that owns the nib file. The object doesn't exist until your program launches so you have no control over the object itself, but you can specify the class of the file's owner.

- 1) Select the File's Owner object from the nib file window.
- 2) If the information panel is not open, open it by choosing Tools > Show Info.
- 3) Select Custom Class from the pop-up menu at the top of the panel, and a large list of classes will appear.
- 4) Choose the class you want from the list.

For the `MainMenu.nib` file Xcode creates for Cocoa application projects, the default File's Owner is the `NSApplication` class. For document-based applications, the default File's Owner is the subclass of `NSDocument` you create.

## First Responder

The *first responder* is the first item to receive events the application receives. Normally the user decides the first responder. Suppose a window has a text field where the user types in text. If the user clicks on the text field with the mouse, the text field becomes the first responder. When the user starts typing, the text field handles the keyboard events and fills the field with the text the user types.

Cocoa windows have an outlet `initialFirstResponder` that lets you specify the initial first responder. Let's say you have a window where the user types in personal information such as name, address, and telephone number. Each piece of information has its own text field for the user to enter the information. In this case you don't want the user to have to explicitly set the first responder by selecting a text field with the mouse. The user should be able to enter his or her name immediately. To let the user start typing the name, set the initial first responder to the text field where the user enters the name.

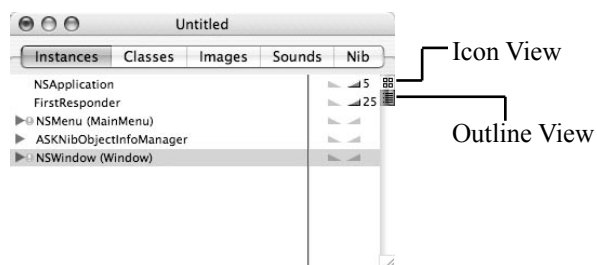
- 1) Make a connection from the window to the text field.
- 2) Set the outlet to `initialFirstResponder`.

## Outline View

On top of the scroll bar at the right edge of the window, you can see two tiny buttons. Clicking the top button places the nib file window in icon view, where each instance has its own icon. The icon view is the default view. Clicking the second button places the nib file window in outline view, which you can see in Figure 3.12. Looking at the figure, you can see two columns. The first column has an entry for each instance. Notice how the menu and window entries show the Cocoa class names. If an entry contains additional items, such as the main menu containing the menus on the menu bars and their menu items, it will have a disclosure triangle next to it. Clicking the triangle will display the additional items in the nib file window.

**Figure 3.12**

Outline view for a Cocoa nib file.



The second column has two triangles for each entry. The first triangle lists the number of outgoing connections and the second triangle lists the number of incoming connections for this item. If a triangle is dim, it means there are no connections. An undimmed triangle with no number means the item has one connection. A number next to the triangle indicates the number of connections.

Clicking a triangle for an item with connections will display the connections, as you can see in Figure 3.13. There will be an entry in the window for each connection to the left of the triangles. Each connection will have an icon next to it. Actions have an icon of a circle with a cross inside it, and outlets have an icon that looks like an electrical outlet (how clever). Next to the icon is the name of the action or outlet. If the connection involves a menu item, the menu will open in Interface Builder. Connections involving controls will show the connection.

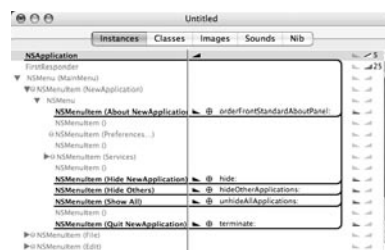
You can use the outline view to make connections between objects. It works similarly to the method I described in the “Making Connections” section earlier in the chapter. Hold down the Control key and select the item you want to start the connection. Drag the mouse to the item in the outline view that should receive the connection. The information panel will open, allowing you to select the action or outlet for the connection.

## Classes Tab

The Classes tab lets you navigate the Cocoa classes and examine each class’s outlets and actions. Use the Classes tab to create subclasses, add outlets, add actions, create instances of classes, and generate source code files.

Initially classes appear in column view. The column view groups the classes hierarchically. Moving left to right in the column view moves you down the hierarchy. The leftmost pane shows `NSObject`, which is the base class of the Cocoa class library. To the right of `NSObject` are the classes that inherit from `NSObject`. If any of these classes have subclasses, a triangle appears next to the class name. Selecting the class shows its subclasses.

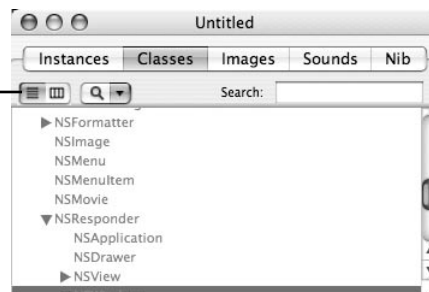
Clicking the leftmost button below the Instances tab changes the class view to outline view, which you can see in Figure 3.14. The outline view places all the classes in the window. The outline view groups the classes hierarchically, using indentation to indicate levels of the hierarchy. If a class has subclasses, that class has a disclosure triangle next to it. Clicking the disclosure triangle reveals the subclasses.



**Figure 3.13**

Connections in outline view.

Outline (left) and column view buttons.



**Figure 3.14**

Column view for Cocoa classes.

## Images Tab

The Images tab shows the images that are part of the nib file. Initially the nib file has one image, the default Mac OS X application icon.

You may want to have other images, such as a different application icon, in your program and may be wondering how to add them to your program. Apple recommends adding the images to your project in Xcode and discourages you from adding them to the nib file. You can add images of any file type the `NSImage` class supports, including TIFF, JPEG, GIF, PICT, and BMP files. After adding the images to the Xcode project, open the nib file, and the images you added should appear in the nib file window when you select the Images tab.

If you want to be a rebel and defy Apple guidelines by adding images to your nib file:

- 1) Choose Interface Builder > Preferences.
- 2) Click the Editing tab.
- 3) Select the Allow image to be stored inside Nibs checkbox.

## Sounds Tab

The Sounds tab shows the sounds that are part of the nib file. The initial entries are system sounds.

## Nib Tab

The Nib tab lets you specify the Nib file format and the oldest version of Mac OS X you want your application to run on. In Mac OS X 10.2, Apple changed the format of nib files to use keyed archiving instead of sequential archiving. Archiving is the process of converting the objects in a nib file to a format the operating system can save to a file. The objects in a sequential archive must be decoded in the same order they were encoded during archiving. Keyed archiving gives each object its own name, a key. You can request an object by its name, giving you more flexibility than sequential archiving.

What the nib format changes mean for you is nib files that use keyed archiving will not load on versions of Mac OS X earlier than 10.2. You have the option of saving nib files in the old format, the new format, or both. Saving nib files in the old format gives you the widest compatibility. Save your files in the new format if you don't care about old versions of Mac OS X. Saving your files in both formats gives you keyed archiving on newer versions of Mac OS X and sequential archiving on older versions. If you use interface elements that were available before Mac OS X 10.2, everything will work fine. However, interface elements introduced later, such as the spinning progress indicator, will not appear for users running old versions of Mac OS X.

The Oldest Target pop-up menu lets you specify the oldest version of Mac OS X your nib file will work on. If it reports no incompatibilities, you're fine. If it reports incompatibilities (Try Mac OS X 10.0 as the oldest target), a Show button appears in the window that lets you see the problems. Choosing File > Compatibility Checking will do the same testing for you.

Selecting the Use text archive format checkbox tells Interface Builder to save the nib file as an XML file instead of a binary file. XML nib files work better with version control systems. Because XML nib files are text files, they're easier to compare with earlier versions. The disadvantage of saving a nib file as XML is a larger file size.

## Creating Cocoa Nib Files

If you don't have many windows or dialog boxes in your application, you can get away with using only the nib files Xcode creates when you create a new project. Should you find yourself creating lots of new windows and dialog boxes, creating multiple nib files is a good idea. You can even go so far as to create a separate nib file for each window. Having each window in its own nib file has two advantages. First, you can use the window in multiple projects; just add the nib file to each project. Second, you can save memory. If you have every window in one nib file, they all load into memory when the nib file loads. By having one nib file per window, each window doesn't move into memory until the program needs it.

To create your own nib files, choose File > New. A window opens that contains the available nib file types. Select the type of nib file you want to make and click the New button to create the file. You can create the following nib files for a Cocoa application:

- Application
- Empty
- Attention Panel
- IB Inspector
- IB Palette
- Human Interface Guideline (HIG) templates

An application nib file is identical to the `MainMenu.nib` file Xcode creates for a Cocoa (not a document-based) application. Use an application nib file to create the user interface before creating an Xcode project.

An empty nib file has no windows or menus. It's a blank canvas for you to create your own windows and dialog boxes.

An attention panel nib file contains a panel. Attention panels alert the user when he or she does something that may cause problems. In Interface Builder if you close the nib file window before saving the changes you made, an attention panel appears, asking you if you want to save the file before closing.

Use the IB Inspector and IB Palette nib files when you want to add your own palettes to the built-in Interface Builder palettes. A *palette* is a group of classes that other developers can use in Interface Builder. Views are the most common items you would create a palette for. The IB Palette nib file creates the palette so other developers can use your classes. The IB Inspector nib file creates an inspector panel so other developers can change the item's attributes in Interface Builder.

Instead of giving you a blank window, the HIG templates place items in the window for you. They provide a starting point for building your user interfaces.

## Creating User Interfaces for Carbon Programs

If you read the whole chapter up to this point, you may be a little disappointed with using Interface Builder for Carbon programs. Interface Builder is strictly for building user interfaces in Carbon. You can't create source code, connect objects, or use bindings. On the bright side, having fewer features makes Interface Builder easier to use for Carbon programs.

Before you start building your program's user interface, read Apple's Human Interface Guidelines. Mac users have expectations on how an application should behave, and they expect your application to behave similarly. You can read the guidelines in Xcode's documentation window by selecting the Human Interface Guide bookmark in the window.

### Laying out the Interface

The nib file for a Carbon application gives you a menu bar and a blank window. To add user interface elements to the window, use the palettes window. The palettes window should be open, but if it's not, choose Tools > Palettes > Show Palettes to open the window. Interface Builder places the user interface items for a Carbon application in six palettes.

- Menus palette
- Controls palette
- Enhanced controls palette
- Browsers and tab palette
- Windows palette
- Text based controls palette

Placing user-interface items is simple. Select a palette by clicking its toolbar button, and select an item from the palette. For all items except the ones in the windows palette and menus palette, drag the item to where you want it to appear in the window. You must drag the items in the windows palette to the nib file window. Refer to the section "Working with Menus" for a detailed explanation on using the menus palette.

### Menus Palette

The menus palette contains the menus and menu items you can add to your application's menu bar. The palette contains the following items:

- Application menu
- File menu
- Edit menu
- Window menu
- Submenus
- Menu items
- Menu separators
- Menu

The menu bar Xcode creates when you create a Carbon application includes the Application, File, Edit, and Window menus so you shouldn't need to explicitly add them. Why are the Application, File, Edit, and Window menus in the palette? There are two reasons. First, you can create an empty nib file and add a menu bar to it. In this case the menus in the menus palette would come in handy. Second, to add new menus to the menu bar, you use the menus in the palette and rename them to suit your purpose. I recommend using the Application menu as the base for new menus. It has only one menu item, simplifying the process of adding menu items.

Submenus work with hierarchical menus. Choose File > Open Recent to see an example of a hierarchical menu. The submenu contains the most recent files you opened in Interface Builder. Menu separators group related items in a menu. If you have a lot of items in a menu, use menu separators to make finding an item in the menu easier.

Dock menus and contextual menus use the Menu object. A Dock menu appears when you click a Dock icon and hold the mouse button. For each running application the Dock menu has items to quit and hide the application as well as items for each open window in the application. Use the Menu object to create a custom Dock menu for your program. Contextual menus work with user interface elements. The menu opens when the user control-clicks the element. The contextual menu contains common commands for the user interface element.

## **Controls Palette**

The controls palette contains the most commonly used controls.

- Buttons. There are both bevel and oval buttons.
- Checkboxes.
- Radio buttons.
- Scroll bars.
- Image wells, which are targets for a user to drag a picture.
- Relevance bars, which measure the relevance of search results.
- Progress indicators.
- Pop-up buttons, which contain pop-up menus.
- Disclosure buttons.
- Segmented views.

Disclosure buttons expand a dialog box to provide additional options. Many applications include disclosure buttons in Save File dialog boxes. The application is set to save the file in a default location. Clicking the disclosure button lets the user choose a location to save the file. Apple added the disclosure button control to Interface Builder 2.5, the version that ships with Mac OS X 10.4. Earlier versions of Mac OS X do not support Interface Builder's disclosure button.

A segmented view is a control that is divided into multiple segments. The back and forward buttons at the top of Finder windows demonstrate the use of segmented views. The buttons are the two segments in the segmented view. Apple added the segmented view to Interface Builder 2.5, the version that ships with Mac OS X 10.4. Earlier versions of Mac OS X do not support Interface Builder's segmented view.

## **Enhanced Controls Palette**

The enhanced controls palette contains additional controls.

- Sliders.
- Separators.
- Round buttons.
- Group boxes.
- Pictures, which display images in a window.
- Disclosure triangles.
- Pop-up arrows, which you would use to display a menu when the user clicks the arrow.
- Little arrows, which increment and decrement values.
- Icons.
- Custom controls.
- User panes, whose main use is to embed controls.

- HViews, which you should use to create custom views.
- HImageViews, which display images in a window.
- Movie views, which allow you to display QuickTime movies.
- Placards.
- Window headers.

Group boxes divide a window into distinct areas. If your window has lots of user interface elements, using group boxes makes your interface easier to comprehend. If you're going to use a group box, start by creating the group box. Drag the controls into the group box.

Both pictures and HImageViews display images in a window. HImageViews are easier to work with, but HImageViews only work on Mac OS X 10.2 and later. Use pictures to support earlier versions of Mac OS X.

HViews are versatile. They can replace custom controls and user panes. Apple recommends using HViews instead of custom controls in Mac OS X. The downside of HViews is that Apple introduced them in Mac OS X 10.2. If you want to support earlier versions of Mac OS X, you'll need custom controls and user panes.

A placard displays information at the bottom of the window. Adobe Acrobat Reader uses a placard to display the page size of PDF documents. Placards can also include a pop-up menu. A window header displays information, and it resides under the window's title bar.

Apple added the movie view, placard, and window header controls in Interface Builder 2.5, the version that ships with Mac OS X 10.4. Earlier versions of Mac OS X do not support these controls.

## Browsers and Tab Palette

The browsers and tab palette contains data browsers and a tab view. There are two data browsers: list and column view. List view data browsers displays rows and columns of data. Column view data browsers display lists of data. If the data is not hierarchical, the browser has one column. If the data is hierarchical, the browser can display one column for each level of the hierarchy.

To see the difference between list view and column view data browsers, go into the Finder. Choosing View > As List tells the Finder to use a list view data browser in Finder windows. Choosing View > As Columns tells the Finder to use a column view data browser in Finder windows. When you view as list, each folder and file has four columns of information: Name, Date Modified, Size, and Kind. You can sort the folders and files by any of these columns. When you view as column, there are no Date Modified, Size, and Kind columns. The files and folders are sorted alphabetically. Selecting a folder shows the folder's contents in the next column.

Tab views allow you to show multiple pages of information in a single window. Selecting a tab makes that tab's page appear in the window. Interface Builder's nib window is an example of a tab view. It has three tabs for Carbon programs: Instances, Images, and Nib.

The initial tab view has two tabs when you place it in a window. To add more tabs to the tab view:

- 1) Select the tab view.
- 2) Open the tab view's information panel by choosing Tools > Show Info.
- 3) Select Attributes from the pop-up menu at the top of the information panel.
- 4) Enter the number of tabs you want in the Number of Items text field.

To place controls in a particular tab, select the tab. Drag the controls you want to the tab's view. Repeat for each tab.

## Windows Palette

The windows palette contains the windows you can add to a Carbon application. It contains the following items:

- Document window
- Movable modal window

Use a movable modal window for movable modal dialog boxes. Movable modal dialog boxes keep the user from doing anything until he or she closes the dialog box. Movable modal dialog boxes do not have close, minimize, and zoom buttons at the top of the window. They contain OK and Cancel buttons inside the dialog to close the dialog box.

## Text Based Controls Palette

The text based controls palette contains the controls that let the user enter text and the controls that display text. It contains the following items:

- EditText, which is a text field for the user to enter text.
- Search field, which lets the user enter text for a search.
- Combo box, which combines a text field with a list.
- ClockDate, which displays the date or time and provides little arrows to change the date or time.
- Text view, which you should use to let the user enter large amounts of text.
- Static text field. Interface Builder supplies three different text sizes for static text fields.

The combo box is available in Mac OS X 10.2 and later. The search field and text view controls are available in Mac OS X 10.3 and later.

## Modifying the Interface

When you lay out your program's user interface, the interface isn't going to look right initially. Buttons, checkboxes, and radio buttons display placeholder text. The default size for views is pretty small. You'll want to make them bigger. This section shows you how to make the modifications your user interface needs.

You can make basic changes easily. Click on a control or a view inside a window to select it. Figure 3.15 shows what a selected item looks like. Dragging the selected item changes its position in the window. To resize a control or view, click one of the eight dots bordering the control you selected, hold down the mouse button, then move the mouse.

To change the text in a control, double-click the text, then enter the new text.

Use the information panel to make other changes. Choose Tools > Show Info to open the information panel. Most items in a Carbon nib file have the following items in the pop-up menu at the top of the panel:

- Attributes
- Control
- Size
- Layout
- Help



**Figure 3.15**

Selected item.



## Attributes Panel

The attributes panel is where you set an item's basic attributes, anything not covered by the other four panels. Use the attributes panel to perform the following tasks:

- Give menu items keyboard equivalents.
- Provide titles for windows and controls.
- Set the number of rows and columns in a radio button group.
- Determine the radio button that should be initially selected in a radio button group.
- Determine whether or not a checkbox should be selected initially.
- Set a slider's minimum, maximum, and initial values.
- Set the number of tabs in a tab view.
- Give a text field placeholder text to display initially.
- Make a text field suitable for entering a password. When the user types text in the text field, dots appear in the text field.
- Add a pop-up menu to a bevel button.
- Determine the controls a window should have: close button, minimize button, zoom button, and resize control.

Each item in the Interface Builder palettes has its own set of attributes, which means what you can set in the attributes panel depends on the item.

## Control Panel

Use the control panel, shown in Figure 3.16, to set the following properties for a control:

- The control's size. A control can be either normal sized or mini.
- The control's font style.
- Whether or not the control is initially enabled.
- Whether or not the control is initially visible.
- The control's signature and ID.
- The control's command.
- The control's properties.

Give a control a signature and an ID so your program can identify the control. The signature is a four-character code that identifies the type of control. The ID is an integer. The combination of signature and ID uniquely identifies a control.



**Figure 3.16**

Controls panel.

A command is a special type of event that menus normally use. When you select a menu item, the operating system sends a command event telling your program the menu item you selected. You can assign commands to controls as well as menus. If you assign a command to a button, the operating system sends a command event when you click the button. The control panel has a pop-up menu that contains frequently used commands you can assign to controls.

What do you do if you want to assign a command that's not in the pop-up menu? Choose Other from the pop-up menu, which fills the Command text field with four question marks. Replace the four question marks with the four-character code you want to use to define your command. Your code may not contain all uppercase letters or all lowercase letters; Apple reserves those codes. Declare a variable of type `UInt32` in your source code with the variable's value being the code you entered in Interface Builder. Call the function `SetControlCommandID()` to assign the command to the control.

Control properties let you add custom data to a control. A Core Foundation dictionary holds the properties, with each property containing one piece of data. Versions of Mac OS X earlier than 10.3 do not support the control properties you set in Interface Builder. Call the function `SetControlProperty()` to add custom data on earlier versions of Mac OS X.

Click the Add button to add a property to the dictionary. A property has four fields.

- Creator, which is a four-character code that identifies your application. Apple reserves creator codes with all uppercase and all lowercase letters. If other people are going to use your program, you should register your program's creator code with Apple.
- Tag, which is a four-character code that identifies the property.
- Type, which is the data type. Use the pop-up menu to choose the data type.
- Value, which is the property data. Double-click a property's value to change the value.

Call the function `GetControlProperty()` to retrieve the data stored in a control property.

If a control has child controls, the control panel lets you set the tab order of the child controls. Click the Tab Order tab to set the tab order. Clicking the Autocalculate button calculates the tab order for you. Even though the control panel is for controls, Interface Builder lets you use the control panel to set the tab order for the controls in a window. Setting the tab order is the only thing windows use the control panel for.

## **Size Panel**

The size panel lets you change an item's position and size. Selecting the item and using the mouse to move and resize it is usually easier than using the size panel. The size panel is good for making smaller changes.

## **Layout Panel**

The layout panel determines what happens when an element's parent resizes. Initially there are no bindings to the parent, which means the element stays the same size. Staying the same size is good for controls like buttons, checkboxes, and sliders.

Views normally should change size when the window resizes. There are four pop-up menus for each part of the element's rectangle. Use the pop-up menus to have an element change size when its parent changes size.

## Help Panel

The Help text box is where you enter the tool tip help. When the user places the mouse cursor over the element, the tool tip help appears.

The Extended Help text box lets you enter more detailed help about a user interface element. When the user places the mouse cursor over the element and holds down the Command key, the extended help appears.

## Working with Menus

Every Mac OS X program with a GUI uses menus. Menus work differently than other user interface elements. You don't drag menu items to a window or to the nib file window. Because menus are so prevalent in Mac OS X programs and require special handling, I've placed the menu building material in its own section.

### Adding Menus to the Menu Bar

To add menus to your application's menu bar, the menu bar must be open in Interface Builder. Double-click the MenuBar item in the nib file window to open the menu bar. Select a menu from the menus palette and drag it to the menu bar to add the menu. If the menu isn't in the right location, select it from the menu bar and drag the menu to the proper location.

Looking at the Interface Builder menu bar, you can see several menus — Classes, Layout, and Tools — that aren't in the menus palette. How do you add menus that aren't in the palette to the menu bar? Drag one of the menus from the menus palette to the menu bar, and double-click the menu bar to change the menu name to what you want. From there you can add, remove, and modify the menu items in your new menu to reflect your application's needs.

### Adding Menu Items

Adding items to a menu is simple. Choose the entry in the menus palette with the name Item (or the blank entry to add a menu separator). Drag the item to the menu in the menu bar where you want to place the menu item. Dragging the items to the menu opens the menu and shows you the items in the menu. Drag the item to where you want it to be in the menu. If you're adding an item to a pop-up menu, double-click the menu to show the menu's existing items before adding the item.

The menu item you just added has the name Item, which is most likely not the name you want. Double-click the item to change the menu item's text. You can also change the names of existing menu items. Remember from the previous section that you can drag a menu from the menus palette and change its name to make it your own menu. If you do this, you'll want to change the menu items as well.

A special menu item is a submenu, a menu item that contains additional menu items. You can see an example of a submenu in Interface Builder's File menu. The menu item Open Recent is a submenu with the additional items being the nib files you opened most recently. Adding a submenu is the same as adding an ordinary menu item. Drag the submenu to the menu. After you add the submenu to the menu, you can add items to the submenu.

Many standard Mac menu items have keyboard equivalents. Pressing Command-Q is the equivalent of choosing Application > Quit. To give a menu item a keyboard equivalent:

- 1) Select the menu item.
- 2) Choose Tools > Show Info to open the menu item's information panel.
- 3) Choose Attributes from the pop-up menu at the top of the information panel.
- 4) Enter the keyboard equivalent in the Key Equivalent text field. There's a pop-up menu for special keys like the function keys and the Esc key.
- 5) If your keyboard equivalent uses modifier keys, select the appropriate Key Modifier checkboxes. There are four checkboxes running from left to right: Command, Shift, Option, and Control.

To delete a menu item, select it and choose Edit > Delete (or press the Delete key). Deleting menus from the menu bar works the same way. Select the menu from the menu bar and choose Edit > Delete to remove the menu from the menu bar along with all the menu's items.

### **Setting a Menu Item's Command**

Each menu item in a Carbon program has a menu command to identify it. When the user selects a menu item, your application receives an event with the menu command the user selected. A menu command consists of four characters. To set a command for a menu item:

- 1) Select the item.
- 2) Open the information panel by choosing Tools > Show Info.
- 3) Choose Attributes from the information panel's pop-up menu.
- 4) The pop-up menu next to the Command text field contains the Apple-supplied commands for the most common menu items. Choose a command from the menu.

What do you do if you create a menu item Apple didn't supply a command for? Choose Other from the pop-up menu next to the Command text field, which fills the Command text field with four question marks. Replace the four question marks with the four-character code you want to use to define your menu command. Your code may not contain all uppercase letters or all lowercase letters; Apple reserves those codes. Declare a variable of type `MenuCommand` in your source code with the variable's value being the code you entered in Interface Builder.

### **Creating a Dock Menu**

Every application in the Dock has a menu the user can display by control-clicking the application's icon. If the application is running, the Dock menu has items to quit and hide the application as well as items for each open window in the application. These items appear automatically for your program without any effort on your program. Creating a Dock menu for your own program with custom menu items requires the following steps:

- 1) Drag a Menu object to the nib file window.
- 2) Add the items you want to add to the menu. You only have to add the custom items that aren't part of the normal Dock menu.
- 3) Add the command for each menu item.
- 4) Add the key `AppleDockMenu` to your program's `info.plist` file. The value for this key is the name of the nib file. Skip the `.nib` extension when entering the nib file name.

## Creating a Contextual Menu

Contextual menus work with user interface elements. When the user control-clicks the element, the contextual menu appears. The contextual menu contains frequently used commands. If you're writing a text editor and the user selects some text, a contextual menu for the text view would have items for cutting, copying, and pasting text. The point of contextual menus is to keep the user from having to move the mouse cursor to the menu bar to perform a menu command. Creating a contextual menu requires the following steps:

- 1) Drag a menu object to the nib file window.
- 2) Add the items you want to add to the menu.
- 3) Add the command for each menu item.

Contextual menus are more difficult to implement in Carbon. In Cocoa you make a connection between the user interface element and the contextual menu. Carbon requires you to write the following code to implement a contextual menu:

- 1) Call the function `CreateMenuFromNib()` to load the contextual menu into memory.
- 2) Create an Apple Event descriptor describing the item you're creating the contextual menu for.
- 3) You must write code to handle the Carbon events `kEventWindowContextualMenuSelect` and `kEventControlContextualMenuClick`. These events handle the user control-clicking to open the contextual menu.
- 4) Call the function `ContextualMenuSelect()` to display the contextual menu.

Call the function `AECreatedesc()` to create an Apple Event descriptor. The first parameter to `AECreatedesc()` is a four-character code, which for a contextual menu is the code identifying the item containing the menu. The code should match the signature you gave the item in Interface Builder.

## The Nib File Window for Carbon Applications

I've spent a lot of time covering the objects that make up a nib file: windows, menus and controls. The nib file window is where you look at the nib file as a whole instead of looking at its individual pieces. The nib file window for Carbon programs has three tabs.

- Instances tab
- Images tab
- Nib tab

### Instances Tab

The Instances tab is less cluttered for a Carbon nib file than a Cocoa nib file. A Carbon application's instances tab contains two items: the menu bar and a window. If you add windows and contextual menus to your interface, the instances area shows what you added.

Initially Carbon nib files show the instances in icon view. Right above the scroll bar at the right edge of the window are two tiny buttons. Clicking the lower one will change the view to an outline view. Figure 3.17 shows what the outline view looks like. If an instance has additional items in it, the instance will have a disclosure triangle next to it. Click the triangle to reveal the additional items.

## Images Tab

The images tab shows the images in the nib file. To have images from your program appear, close the nib file. Open your Carbon project in Xcode. Add compiled resource files with the extension `.rsrc` (The images won't appear without the `.rsrc` extension). Open the nib file and click the Images tab. The picture (PICT) and icon (icns, cicc, and ICON) resources in the resource files will appear in the nib file window.

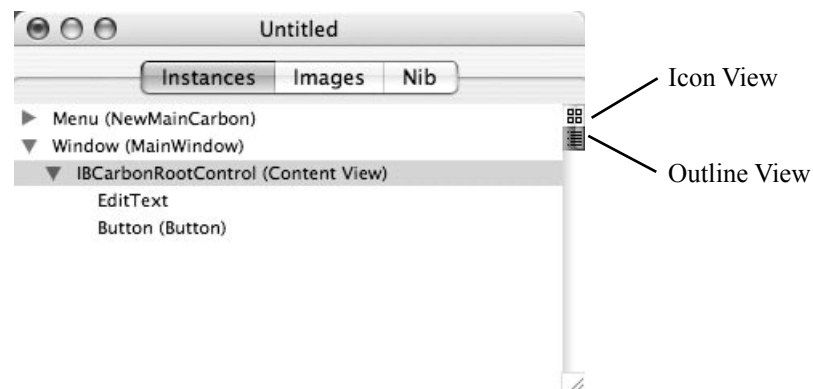
## Nib Tab

For Carbon nib files, the nib tab lets you specify the oldest version of Mac OS X that will work with the nib file. If the nib file is incompatible with an old version of OS X, the number of incompatibilities will appear at the bottom of the window, and a Show button will appear to provide more information about each incompatibility.

## Importing Resource Files

Back in the old days before Mac OS X, Mac developers used resource editors like ResEdit and Resorcerer to visually create user interfaces for their programs. With the move to Mac OS X, Apple replaced their resource editor, ResEdit, with Interface Builder. Since the main reason for developing the Carbon API was to allow Mac developers to move their code to OS X, Interface Builder provides an option for Carbon developers to create nib files out of their old resource files. Interface Builder can convert the following resources from a resource file:

- Menu bars.
- Menus.
- Windows.
- Dialog boxes.
- Dialog item lists, which contain the controls that appear inside a dialog box.
- Controls.



**Figure 3.17**

Outline view for Carbon nib file.

To import a resource file, choose File > Import > Import Resource File. An Open File dialog box appears, asking you to select a resource file to import. After selecting a resource file, a dialog box like Figure 3.18 opens, asking you which items in the resource file you want to import. After selecting the items you want to import, click the Import button. Interface Builder will create a new nib file with all the user interface items you imported from the resource file.

## Creating Carbon Nib Files

If you don't have many windows or dialog boxes in your application, you can get away with using only the nib files Xcode creates when you create a new project. Should you find yourself creating lots of new windows and dialog boxes, using multiple nib files is a good idea. You can even go so far as to create a separate nib file for each window. Having each window in its own nib file has two advantages. First, you can use the window in multiple projects; just add the nib file to each project. Second, you can save memory. If you have every window in one nib file, they all load into memory when the nib file loads. By having one nib file per window, each window doesn't move into memory until the program needs it.

To create your own nib files, choose File > New. A window opens that contains the available nib file types. Select the type of nib file you want to make and click the New button to create the file. You can create the following nib files for a Carbon application:

- Main window with menu bar
- Empty
- Window
- Dialog
- Menu Bar
- Human Interface Guideline (HIG) templates

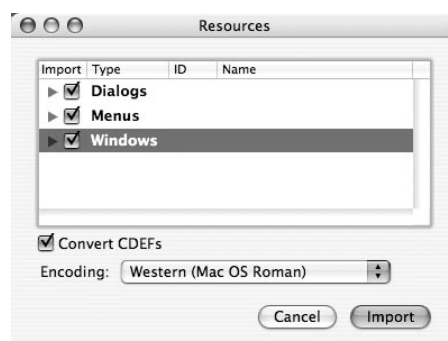
The Main window with menu bar nib file is identical to the `main.nib` file Xcode creates for a Carbon application. Use an application nib file to create the user interface before creating an Xcode project.

The empty nib file has no windows or menus. It's a blank canvas for you to add windows and dialog boxes.

The window nib file contains one document window. The dialog nib file contains one movable modal dialog box.

The menu bar nib file contains a menu bar with the Application, File, Edit, and Window menus. Using a menu bar nib file takes nib file separation to the extreme, separating the menu bar from the windows and dialog boxes in your Carbon program.

Instead of giving you a blank window, the HIG templates place items in the window for you. They provide a starting point for building your user interfaces.



**Figure 3.18**

Import resources dialog box.

# Chapter 4

## Sampler

When you start writing a program, you spend most of your time in Xcode and Interface Builder, the tools I covered in the first three chapters. Once you get your program running, you can move on to the performance tools that come with the Xcode Tools. With these performance tools you can find problems such as slow spots in your code, excessive memory usage, and memory leaks. One of the easiest performance tools to use is Sampler, which can perform the following tasks:

- Find the functions where your program spends the most time.
- Examine the memory allocations your program makes.
- Examine the function calls made when you call a function in one of Apple's frameworks.

### Call Stacks

To be able to understand the information Sampler provides, you must know about call stacks. A *call stack* is a list of an application's active functions, the functions the application called that haven't returned yet. Figure 4.1 shows an example of a call stack. When your program calls a function, the operating system places the function at the top of the call stack. The operating system removes the function from the call stack when the function returns.

Call stacks include functions your program doesn't directly call, which makes the stacks taller than you would initially believe. When your program launches, the operating system calls a series of low-level functions. Some of these functions stay on the call stack until the user quits the program. Additionally, the functions in Apple's frameworks make calls to other functions. When you call a Cocoa method, the method calls a series of functions. The series of functions appears on the call stack even though you didn't directly call any of those functions in your code.

As your program runs, Sampler periodically examines the call stack and records the functions on the stack. Sampler keeps a list of every function appearing at least once on the call stack. For each function on the list, Sampler keeps track of the number of times it found the function on the call stack. When Sampler stops recording, it displays the call graph for the recording session. The call graph contains a list of every function making at least one appearance on the call stack along with the number of times each function appeared on the stack. Sampler stores the call graph in a top-down hierarchy. At the top of the hierarchy is the first function your program calls. The second level of the hierarchy contains the functions the first function called. The third level contains the functions the second level routines called and so on.

Function A (current function)
Function B (called A)
Function C (called B)
main
start (calls main when program starts)

**Figure 4.1**

Sample call stack. Each function in the stack called the one above it.



## Running Sampler

After launching Sampler, your first step depends on the application you want to sample. If the application is running, choose File > Attach. A dialog box opens with a list of running programs for you to choose.

If the application you want to sample is not running, choose File > New. A dialog box like Figure 4.2 opens, asking you for a program to sample. Click the Set button next to the Executable combo box. An Open File dialog box opens for you to choose the program.

The Arguments combo box lets you supply any runtime arguments your application needs to run. Most Mac programs don't need to use the Arguments combo box; only programs that run from the command line use runtime arguments.

The Working Dir combo box lets you specify a default directory to read and write files. Initially the working directory is the /tmp directory. Most Mac programs use Open File and Save File dialog boxes instead of working directories. Unless you're writing a command-line application, you won't have to worry about setting a working directory.

### NOTE

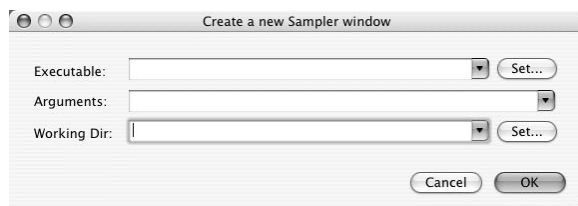
You can also use Xcode to run your application in Sampler. Open your project in Xcode and choose Debug > Launch using Performance Tool > Sampler. The dialog box shown in Figure 4.2 will open, but Sampler fills the Executable and Working Dir combo boxes with your application's information.

## Profiling Your Program

After telling Sampler the program you want to profile, a window like Figure 4.3 opens. There's not much to see because you haven't started sampling. Sampler has three ways to watch your program.

- Timed samples. Sampler inspects the call stack at fixed time intervals, usually multiple times per second.
- Memory allocations. Sampler looks at the call stack every time your program allocates memory.
- Function calls. You supply a list of functions to watch for. Sampler examines the call stack every time your program calls a function on the list.

The timed samples method is the best way to run Sampler for profiling purposes. Make sure the Watch for pop-up menu in the upper left corner says Timed Samples.



**Figure 4.2**

Dialog box to run Sampler on a program that is not currently running.

Use the drawer connected to the Sampler window to set Sampler's sampling rate. The default value is 20 milliseconds, which means Sampler looks at the call stack 50 times per second. You can sample at a rate up to 1 millisecond (1000 samples per second). A higher sampling rate gives you a more accurate picture of your program's performance at the cost of making your program run slower when Sampler samples. The default sampling rate of 50 times per second should be sufficient in most cases.

After choosing a sampling rate, start sampling by clicking the Launch and Record button (If your application is already running, the button will say Start Recording). Your application will launch, and you should run it the way a typical user would. When you're finished sampling, keep your application running, and click the Stop Recording button in Sampler. If you quit your program before clicking the Stop Recording button, no data will appear in the Sampler window.

If you want to wait until you reach a certain area of your program to start sampling, click the Launch button, which will launch your application. Click the Start Recording button when you're ready to sample.

## Examining the Results

When you click the Stop Sampling button, the sampled data fills the window. By default the window shows the trace view, but it isn't the best view for finding the areas of your code where the computer spends the most time. Click the Browser tab and the window will switch to the browser view, which you can see in Figure 4.4. The browser view consists of two parts: the call graph browser and the call stack pane.

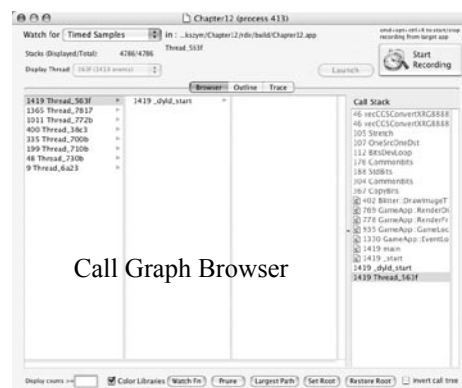
The call graph browser lets you navigate the call stacks in your program. Sampler organizes the call graph in a top-down hierarchy. Moving left to right in the call graph browser moves you down the call graph hierarchy. If a function has a triangle next to it in the browser, that function calls additional functions.

Initially the bottom of the call stack pane contains the first function (function A) your application called. On top of it is function B, the routine that A called the most. On top of that is function C, the routine that B called the most, and so on. The call stack pane gives you a way to quickly examine frequently called functions.



**Figure 4.3**

Sampler window before you start recording.



**Figure 4.4**

Sampler browser view after recording.

## Threads

The first piece of information the call graph browser displays is a list of the threads in your program. The list runs down the left column of the call graph browser. You can see an example of a thread list in Figure 4.4. A *thread* is a piece of a program that can run independently from the rest of the program. Every Mac program you write has at least one thread, the program itself. The operating system can add threads to your program as it runs to improve performance. The program sampled in Figure 4.4 has eight threads, even though I wrote code for only one thread. By moving a large task into its own thread, a program can work on the task without disrupting the rest of the program.

Suppose you're in a situation similar to the one shown in Figure 4.4. You wrote a single threaded program, but Sampler shows eight threads in the call graph browser. How do you know which thread contains the code you wrote? Select a thread from the call graph browser and examine the call stack pane. Some of your functions will appear in the call stack pane if you select the thread containing your code.

To look at a function, select it from either the call graph browser or the call stack pane. Selecting a function changes the call graph browser. The middle pane in the call graph browser shows the function you selected. The right pane shows the routines the selected function called. The left pane shows the routines one level above the function you selected. The routine that called the selected function has a gray highlight. Clicking on the scrollbar at the bottom of the call graph browser lets you navigate the call stack.

If the function you select from the call graph browser is not in the call stack pane, Sampler updates the call stack pane to reflect the selection. The function you selected, function S, appears in the call stack pane above the routine that called S. Function T, the function that S called most, appears above function S. Function U, the function that T called most, appears above function T, and so on.

Each function in the call graph browser has a number in front of it. The number tells you the number of times the function appeared on the call stack when Sampler sampled your program. The number of call stack appearances is the only information Sampler supplies about a function, making this information significant. Keep the following points in mind:

- The number of times a function appears on the call stack does not equal the number of times your program called it.
- The number of times a function appears on the call stack has nothing to do with the amount of time your program spends in that function.

The number of times a function appears on the call stack has nothing to do with how many times your program called it. When your C, C++, Java, or Objective C program launches, the operating system calls the `main()` function. `main()` doesn't return until the user quits the program, which means `main()` appears on the call stack during every sample Sampler takes. If Sampler takes 3000 samples, `main()` appears on the call stack 3000 times, even though `main()` was called only once.

The number of times a function appears on the call stack reflects more on how long the program runs than on the amount of time the program spends in the function. Suppose you run your program for one minute and see a function appears on the call stack 100 times. You run the program a second time for five minutes and find that same function appears 500 times. Did the function get five times slower because it appears on the stack 500 times instead of 100? No, the function appears on the call stack more because the program ran longer. The longer a program runs, the more samples Sampler takes, which means more chances for a function to appear on the call stack.

When examining a function, you want to know if the program spends a lot of time in that function. The number of call stack appearances provides a clue, but you must take the information in context. Suppose one of your functions, function A, appears on the call stack 500 times. The information itself, the 500 appearances, tells you nothing. You must look at the function, function B, which called function A. If function B appears on the call stack 5000 times, function A isn't a slow spot; B spends only 10 percent of its time in A. If function B appears on the call stack 550 times, function A could be a slow spot because B spends over 90 percent of its time in A. You would have to look deeper at function B to determine if optimizing A would be worth the effort.

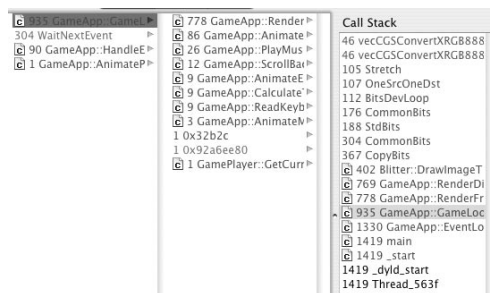
If you select a function in the call graph browser, you will notice the sum of the numbers in the pane to the right of the function equals the number of the function. Figure 4.5 shows an example. The function I selected in Figure 4.5, `GameApp::GameLoop()`, has 935 call stack appearances; you can see it more clearly in the call stack pane. Add the numbers next to the functions in the right pane of Figure 4.5. You should get a sum of 935. If I wanted to make `GameApp::GameLoop()` run faster, I would focus on `GameApp::Render()`, the function that appears on the call stack 778 times. Optimizing any of the other functions `GameApp::GameLoop()` calls would be a waste of time since `GameLoop()` spends about 83 percent of its time in the `Render()` function.

Functions you wrote have an icon next to them in the browser view. C, C++, and Objective C functions have an icon of a sheet of paper with the letter C. Java functions have the letter J in the icon instead of C. Double-clicking a function you wrote opens the function's source code file in Xcode and takes you to the function. You must be running Xcode to be able to open the file.

## Outline View

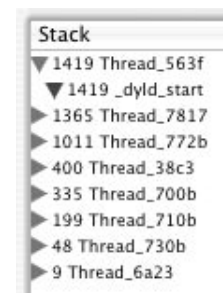
Click the Outline tab to change the view from the browser view to the outline view. The outline view eliminates the call stack pane and switches the call graph browser to a single view of the call stack hierarchy, as you can see in Figure 4.6. If a function calls additional functions, that function will have a disclosure triangle next to it. Clicking the disclosure triangle reveals the routines that function called. The outline view allows you to see more levels of the call stack hierarchy at one time than the browser view, which limits you to three levels.

The outline view provides the same information the browser view does. The number next to a function tells you the number of times the function appears on the call stack. Functions you write are underlined in the outline view. Double-clicking one of these functions opens the function's file in Xcode and take you to the function, provided you have Xcode running.



**Figure 4.5**

The numbers next to the functions in the right pane of the call graph browser add up to the number next to the selected function in the center pane.



**Figure 4.6**

Sampler outline view.

## Trace View

The trace view, which you can open by clicking the Trace tab, provides information on each sample Sampler took during the recording session. Figure 4.7 shows a sample trace view. At the top of the trace view is a graph that plots the call stack depth, the number of functions on the call stack, for each sample. Upward spikes in the graph indicate increased activity in your program. The samples where the spikes occur are the most interesting to examine.

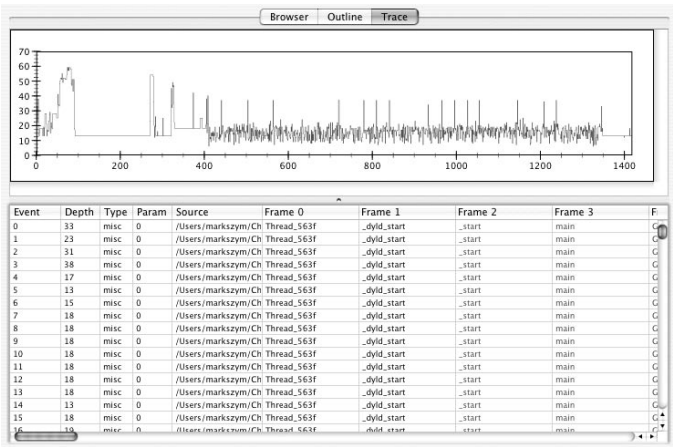
Below the graph is a table providing the following data for each sample:

- Event, which is the sample number. Sample numbers start with zero.
- Depth, which is the number of functions on the call stack during this sample.
- Type, which labels memory allocations and deallocations.
- Param, which contains the amount of memory allocated.
- Source, the line of source code where the program was this sample.
- The call stack. There are many Frame columns with one entry for each function in the call stack. Frame 0 is the bottom of the call stack.

Only memory allocations use the Type and Param columns. You can ignore those columns if you run Sampler using timed samples.

Clicking on a line in the trace view graph will take you to that sample in the table. A green line appears on the graph to tell you where you are.

Initially the trace view graph fits the whole graph on the screen. You may want to focus on a particular group of samples. Selecting an area of the graph will zoom the graph to that area. Holding down the Command key and clicking on the graph reverses the zoom.



**Figure 4.7**  
Sampler trace view.

## Examining Memory Allocations

What do memory allocations have to do with the performance of your application? Mac OS X has virtual memory, using free hard disk space as extra memory. The more memory your application uses, the greater the chances of the operating system having to go to the hard disk, slowing the application.

To examine memory allocations, choose Memory Allocations from the Watch for pop-up menu. When watching memory allocations, Sampler records the call stack each time your program allocates or deallocates memory so there's no need to set a sampling rate. Selecting the Show only active blocks checkbox tells Sampler to ignore memory deallocations. Click the Start Recording button for Sampler to record your program's memory allocations. Run your program and leave it running when you're finished. Click the Update button to see the results.

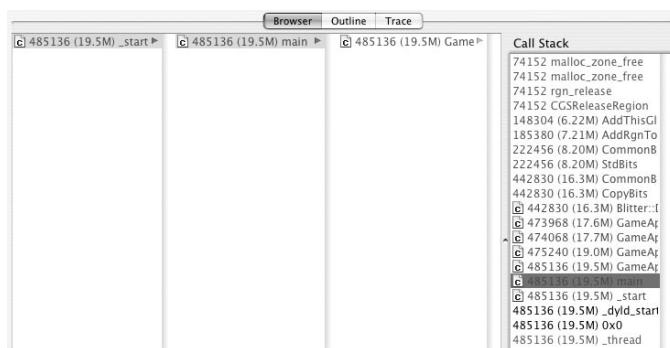
Figure 4.8 shows the browser view for memory allocations; the outline view shows the same information. The number next to each function represents the number of call stack appearances the function made, which is the same information Sampler provides for timed samples. Sampler provides an additional piece of information for each function when watching for memory allocations. It tells you how many bytes of memory your program allocated when the function appeared in the call stack. The number of bytes appears in parentheses next to the function name.

There are two things to be aware of when examining memory allocations. First, Sampler sorts the function calls by the number of times they appear on the call stack. A function that allocates 20 bytes of memory 500 times appears ahead of a function that allocates 20 MB of memory once. Second, the number next to each function does not state how much memory this function allocated. Suppose function A calls function B, which allocates 1 MB of memory. The 1 MB allocation will appear in parentheses for both function A and function B, even though function A didn't make the allocation.

The trace view allows you to examine every memory allocation your program made. The graph measures the depth of the call stack for each allocation, and the table provides information about each allocation. The table has the same information it does for timed samples, but the Type and Param columns mean something for memory allocations. A memory allocation has the value `alloc` in the Type column, and a memory deallocation has the value `dealloc`. The Param column reports the size of the memory allocations in bytes; deallocations have the value 0.

After looking at memory allocations, you can go back to your program without losing what you examined. Clicking the Update button instead of the Stop Recording button tells Sampler to add the new samples to the ones it previously took. As long as you don't click the Stop Recording button, Sampler continues adding samples.

Clicking the Stop Recording button erases all the samples Sampler took. Erasing the samples is the right choice when you're finished with the memory allocation data. By wiping out the old samples, you can go back to your program and focus on the new memory allocations.



**Figure 4.8**

Sampler output for memory allocations.

## Examining Specific Functions

When looking at your program, you may be interested in only certain functions and want to restrict your examination to those functions. You can tell Sampler to record the call stack only when your program calls specified functions. Choose Function Calls from the Watch for pop-up menu. A drawer like Figure 4.9 will attach itself to the Sampler window.

Initially the list of functions to observe is empty. Sampler supplies six sets of functions in the Add Function Group section. The functions in these groups are C standard library functions and Unix system calls, routines that Cocoa and Carbon functions call behind the scenes. Click a group's button to add its functions to the Functions to observe list. Use the Delete button to remove functions you don't care about from the list.

How do you tell Sampler to watch functions you wrote? You have two options. First, you can type the function name in the text field below the Functions to observe list and click the Watch button. Typing the function names is quick, but potentially troublesome because Sampler doesn't know anything about your code. If you make a typing error, Sampler won't detect the error and won't find any calls to the mistyped function.

Second, you can run a timed samples trace. The browser view lists the functions that appear in the call stack. Select a function you want to examine from the browser view and click the Watch Fn button; repeat for every function you want to examine. Choose Function Calls from the Watch for pop-up menu and the Functions to observe list will contain the functions you added with the Watch Fn button. Now you can start recording with the peace of mind that comes with avoiding typographical errors.

Click the Start Recording button to start viewing function calls. Run your application and click the Stop Recording button to examine the results. While your program runs, Sampler records the call stack every time your program calls one of the functions on the Functions to observe list. Sampler displays the same information when watching for function calls as it does when watching for timed samples. The number next to each function in the browser and outline views represents the number of call stack appearances the function made. The trace view lets you examine every sample Sampler took.

Entering a list of functions for Sampler to watch is tedious, especially if you run Sampler multiple times on the same group of functions. Sampler allows you to save a group of functions so you don't have to enter the functions every time you run Sampler. Click the Save button in the drawer to save the functions in the Functions to observe list. Clicking the Load button allows you to restore a saved function list to run new traces.



**Figure 4.9**

The drawer attached to the Sampler window when looking for calls to specific functions.

## Options to Control Sampler's Output

As I've said numerous times this chapter, the number next to each function represents the number of call stack appearances the function made. If you're watching for timed samples, you can display different information next to each function. The drawer attached to the Sampler window has a pop-up menu with the label Display counts as. Sampler initially displays counts as samples, but there are two additional options. First, Sampler can display the number of seconds a function appeared on the call stack. Second, Sampler can display the percentage of time a function appeared on the call stack. How you display counts – samples, seconds, or percent – doesn't matter. Samples, seconds, and percent are just different ways to display the same call stack data.

A suite of controls runs along the bottom of the window that lets you control Sampler's output. Below is the list of controls, running from left to right.

- Display counts text field
- Color Libraries checkbox
- Watch Fn button
- Prune button
- Largest Path button
- Set Root button
- Restore Root button
- Invert call tree checkbox

Enter a number in the Display counts text field and only functions with a sample count greater than the number you type in will appear in the Sampler window. Things can get confusing with the Display counts text field if you display sample counts as seconds or as percents. The Display counts text field deals with sample counts only. Suppose you're displaying sample counts as percents, and you want to look only at functions that appear on the call stack more than 25 percent of the time. If you type 25 in the text field, Sampler will show all functions that appear on the call stack more than 25 times, not the functions that appear on the call stack more than 25 percent of the time.

To display functions that appear on the call stack more than a certain percentage of time, look at the total number of stacks, which appears below the Watch for pop-up menu at the top of the window. Type the total number times the percentage you want in the Display counts text box. If you have 1000 samples and want to see functions that appear more than 20 percent of the time on the call stack, type 200, which is 20 percent of 1000.

Selecting the Color Libraries checkbox colors the text of a function by the code library it belongs to. Coloring the text makes finding the functions you wrote easier; your functions appear in a different color than functions in external libraries like Carbon, Cocoa, and Core Foundation. If you have lots of classes in your code, Sampler may give each class its own text color. There's no way to predict the colors Sampler will use. I ran Sampler on one of my programs four different times and had my functions colored black, green, cyan, and gray.

Selecting a function from the call graph browser and clicking the Watch Fn (Fn is an abbreviation for function) button adds the function to the list of functions Sampler's observing. You can see the list by choosing Function Calls from the Watch for pop-up menu.

The call trees the call graph browser shows can become enormous. Sometimes there are functions you don't want to examine. Selecting a function from the Sampler window and clicking the Prune button temporarily hides the function in the call tree. Pruning a function is similar to the premise of the movie *It's a Wonderful Life*. You see how the call stacks look if the pruned function never existed. The functions above the pruned function have lower call stack appearances while the routines the pruned function called disappear.



When you prune a function, it appears in the filter panel, which you can see in Figure 4.10. Choose Tools > Filter Panel to view the filter panel. You can restore a pruned function by selecting it from the filter panel and clicking the Restore button. Clicking the Restore all button restores all the pruned functions, bringing the call tree back to its original state.

Clicking the Largest Path button fills the call graph browser with the path from the start of your program to the leaf function with the most call stack appearances. You should prune any threads the operating system created before clicking the Largest Path button. These threads usually call one function repeatedly, giving the leaf function a high number of call stack appearances. When you click the Largest Path button, one of these extra threads is most likely to appear in the call graph browser. Use the Restore Root button to leave the largest path view.

When examining your application, you must wade through layers of function calls to find a function you wrote and possibly wade through even more levels to find an interesting function. By selecting a function and clicking the Set Root button, you set the top of the call stack hierarchy to the selected function. Using the Set Root button hides the boring top layers of the call stack hierarchy, allowing you to focus on the code you wrote. Clicking the Restore Root function restores the original root in the call tree. The trace view does not use the Set Root and Restore Root functions.

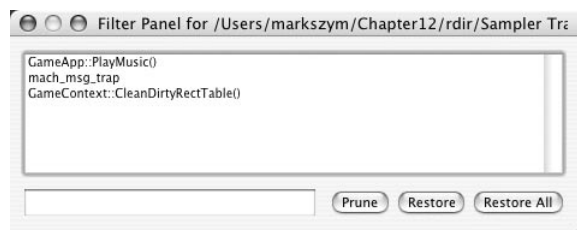
By default the call graph browser starts at the top of the call stack hierarchy. Selecting the Invert call tree checkbox reverses the call graph browser. The browser starts at the bottom of the call stack hierarchy with the leaf functions, the functions in your program that did not call other functions. Moving left to right in the call graph browser moves you up the hierarchy. Inverting the call tree makes it easy to see how your programs call the leaf functions, especially if you use the browser view. Selecting a leaf function in the call graph browser fills the call stack pane with the path from the leaf function to the start of your program. The ability to see the complete path makes inverting the call tree helpful when watching memory allocations.

## Generating Reports

After examining your program with Sampler, you may want to look at the results later. Sampler provides two ways of saving your results. First, you can save a Sampler session to look at later. Choose File > Save Trace As to save the session. Choose File > Open Trace to view a saved session.

Second, you can use reports to display information about a Sampler session. You can save a report to a text file by choosing File > Save Report or print the report by choosing File > Print. Sampler can generate three reports.

- Call graph report
- Function cross reference
- Library cross reference



**Figure 4.10**

Sampler filter panel.

## Call Graph Report

Choose Tools > Generate Report to generate the call graph report. It contains the complete call graph of your code for timed samples, memory allocations, or function calls, depending on what you were viewing in Sampler. The report looks like the Sampler outline view with all the disclosure triangles expanded. You may want to use the Set Root button to eliminate areas of your program from the report before generating it. Large programs have a lot of indentation in the call graph report, making it hard to read.

The call graph report also contains two pieces of summary information. The first piece has the heading Total number in stack. It contains a list of all functions appearing in five or more places on the call graph. What causes a function to appear in more than one spot on the call graph? When multiple routines call a certain function, that function appears in one place for each routine calling it. The second piece of summary information has the heading Sort by top of stack. It contains a list of all leaf functions your program called five times or more.

## Function Cross Reference

Choose Tools > Function Xref to create the function cross reference. The function cross reference has a listing for every function call your program made. Each listing has three pieces of information.

- Count, the number of call stack appearances.
- Parent, the function making the call.
- Child, the function the parent called. Leaf functions have the thread number as the child.

## Library Cross Reference

Choose Tools > Library Xref to create the library cross reference. The library cross reference contains a listing of the code libraries used for the function calls your program made. Each listing has three pieces of information.

- Count, the number of call stack appearances.
- Parent, the library of the function making the call.
- Child, the library of the function the parent called.

A listing in the library cross reference represents all the calls the functions in the parent library make to the functions in the child library. The two libraries can be the same, such as when a function in your application calls another function in the application. Suppose your program makes calls to QuickTime functions. The library cross reference will have one listing with your program's executable file as the parent and the QuickTime library as the child. The count will be the total number of call stack appearances the QuickTime functions make. If your program calls 15 QuickTime functions and each function appears on the call stack 100 times, the library cross reference reports a count of 1500.

# Chapter 5

## gprof

`gprof` is a profiler, a program that collects information about your program as it runs. Use a profiler to find the functions in your code that are running too slowly so you can speed them up. For each function in your program, `gprof` reports the number of times your program called it, the amount of time your program spent in the function, the routines that called it, and the routines it called.

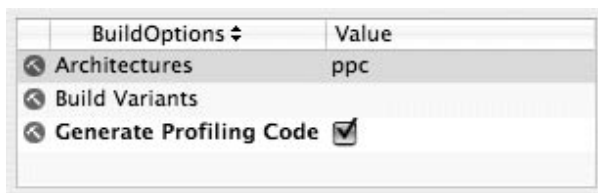
### Generating Profiling Code In Xcode

Before you profile your program with `gprof`, you must set up Xcode to create the profiling information `gprof` requires. Normally your code just executes when running your program. To profile your program it must record statistics about your code — the functions your program calls and the amount of time it spends in each function — as the code executes. You must tell Xcode to generate profiling code that records the statistics about your code as it executes.

- 1) Select the project name from the Groups and Files list.
- 2) Click the Info button to open the project's information panel.
- 3) Click the Build tab in the information panel. Some versions of Xcode have a Styles tab.
- 4) Choose Build Options from the Collections pop-up menu.
- 5) Select the Generate Profiling Code checkbox, which you can see in Figure 5.1.

While you're modifying your project settings, you should change the compiler's optimization level. Turn up the optimization level to match the level you would use when your program is ready for release. Using optimized code provides a more accurate profile. If you profile your program with unoptimized code, you may end up wasting time on code the compiler would optimize for you automatically. To set the optimization level, choose Code Generation from the Collections pop-up menu.

After tinkering with your compiler settings, you must decide on the runtime library you want to use. Initially Xcode uses the standard runtime library. If you profile your program with the standard library, `gprof` generates profiling information only on the functions you wrote. No profiling information will appear for functions your program calls from external frameworks like Cocoa and Carbon. Use the profile runtime library if you want to see profiling information on the functions from external frameworks. To use the profile runtime library:



**Figure 5.1**

Xcode's compiler settings for code generation.

- 1) Select Executables from the Groups and Files list. The list of executable files appears in the window.
- 2) Select the executable name of the program you want to profile.
- 3) Click the Info button to open the information panel for the executable file.
- 4) Click the General tab.
- 5) You should see a pop-up menu with the label Use suffix when loading frameworks (See Figure 5.2). Choose profile from the menu.

## Running gprof

Now that you have set up Xcode to generate profiling code, you must recompile your program so it will record the profiling data when you run the program.

- 1) Clean your project by choosing Build > Clean.
- 2) Build your project by choosing Build > Build and grab a snack while Xcode compiles all the files.
- 3) Run your application the way a user normally would.

After running your application, a file named `gmon.out` appears in the same folder as your application. The `gmon.out` file contains the profiling data recorded when you ran your program. `gprof` uses the `gmon.out` file to generate the profiling report. You can rename the `gmon.out` file if you want, but running `gprof` is easier if you don't rename the file.

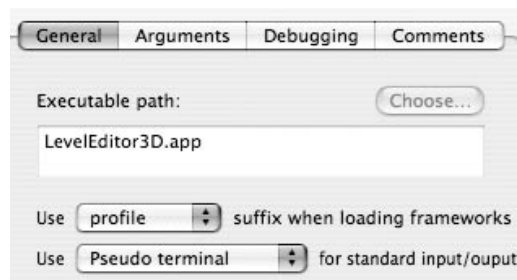
After creating the `gmon.out` file, run `gprof` and generate a profiling report. To run `gprof` you must move from the Mac OS X GUI to the command line by running the Terminal application. Move to the directory where your application resides. Refer to the section “A Command Line Primer” in Chapter 9 for more information about navigating the command line. The command to run `gprof` takes the form.

```
gprof [Options] [Executable File] [Profile Data Files] > [Output File]
```

### NOTE

If the name of your application has spaces in it, such as My Great Program, place quotes around the path name so the operating system treats the path as one argument. Without the quotes the operating system would treat the words My, Great, and Program as separate arguments, and `gprof` wouldn't run.

I want to keep things relatively simple so I'm going to ignore the options right now. *Executable File* is the name of your application.



**Figure 5.2**

Setting the runtime profile library in Xcode.

`gprof` assumes the profile data file's name is `gmon.out`. If you keep the name `gmon.out`, you don't have to supply it to `gprof`. If you rename the `gmon.out` file, you must supply the file's name to `gprof`. When I cover the various `gprof` options later in the chapter, you will learn about other profile data files you can supply to `gprof`.

If you supply an output file, `gprof` writes the profiling information to that file. If you do not supply an output file, the results of the profile appear in the shell window.

### WARNING

If there is a file with the same name as the output file you supply to `gprof`, `gprof` will overwrite the existing file.

Theoretically the simplest way to run `gprof` is to supply only an executable file.

```
gprof ExecutableFileName
```

In this simple case, you supply no options, the profile data file has the name `gmon.out`, and the profiling results appear in the shell window.

The simple case works on Unix systems but not on Mac OS X. The problem running `gprof` on Mac OS X is `gprof` is unaware of OS X's bundle structure. Bundles are directories that appear to the end user as a single file. The executable file resides three directories inside the application bundle.

```
Application Bundle
  Contents
    MacOS
      Executable File
```

If you go to the directory containing the application bundle and run `gprof`.

```
gprof AppTitle.app > results.txt
```

`gprof` won't be able to find the executable file. If you navigate the three directories to the location of the executable file and run `gprof`, it won't be able to find the `gmon.out` file. There's no way to avoid typing a path name. The easiest place to run `gprof` is from the directory containing the application bundle.

```
gprof AppTitle.app/Contents/MacOS/AppTitle > results.txt
```

Where *AppTitle* is the title of your application. By running `gprof` from the directory containing the application bundle, you type only one path name. Assuming you didn't change the name of the `gmon.out` file, running `gprof` the way I did in the previous example creates a text file called `results.txt` in the current directory and writes the profiling report in that file.

When you run `gprof` it can take several minutes to generate the profiling report. During this time the cursor moves to the left side of the shell window and you won't be able to type any commands. You haven't crashed the computer; `gprof` is working. When `gprof` finishes, the cursor will return to normal and you will be able to type commands in the shell window. At this point you can leave the Terminal application and look at the report `gprof` built for you.

## Interpreting gprof's Results

To see the results of your profile, go to your favorite text-editing program and open the file `gprof` made. If you chose not to create an output file, look at your shell window. `gprof` generates two reports, a flat profile and a call graph profile. The call graph profile appears first, but the flat profile is easier to understand so I will start there.

### Flat Profile

The flat profile lists all the functions your program calls. Conveniently, `gprof` sorts them by execution time; the functions where your program spends the most time appear at the top of the list. With the flat profile you can easily find the most time consuming functions in your program. For each function, the flat profile provides seven pieces of information, each in its own column, as you can see in Table 5.1.

If you read Table 5.1 you will see that many columns in the flat profile are blank if `gprof` did not profile a function. Why would `gprof` not profile a function? The most common cause of `gprof` not profiling a function is building your program with the standard runtime library instead of the profile library. `gprof` won't profile calls to functions in external frameworks if you build your program with the standard runtime library. `gprof` should profile every function you write no matter what runtime library you use.

Every program you profile has the functions `moncount()` and `mcount()` in the flat profile. `gprof` calls these functions to do its profiling. Don't worry about the amount of time your program spends in `moncount()` and `mcount()`. They reflect the amount of time `gprof` spent counting. When you look at the flat profile and the call graph profile, you'll see that the total time is different. `moncount()` and `mcount()` are the reason for the total time difference. The functions appear in the flat profile, but not the call graph profile.

**Table 5.1 Flat Profile Columns**

Column	Description
<code>%time</code>	The percentage of the total running time your program spent in the function.
<code>cumulative seconds</code>	Keeps a running total of how much time the program spent in the function and the functions above it in the profile. If the program spent 1.6 seconds in the first function and 1.3 seconds in the second function, the <code>cumulative seconds</code> column for the second function would be 2.9.
<code>self seconds</code>	The number of seconds the program spent in the function.
<code>calls</code>	The number of times your program called the function. This column will be blank if <code>gprof</code> did not profile the function.
<code>self ms/call</code>	The average number of milliseconds your program spent in the function per call. Take the <code>self seconds</code> column, multiply by 1000, and divide by the <code>calls</code> column. This column will be blank if <code>gprof</code> did not profile the function.
<code>total ms/call</code>	The average number of milliseconds your program spent in the function and its descendants per call to the function. The descendants are the routines called by the function, the functions those routines call, and so on. This column will be blank if <code>gprof</code> did not profile the function.
<code>name</code>	The name of the function.

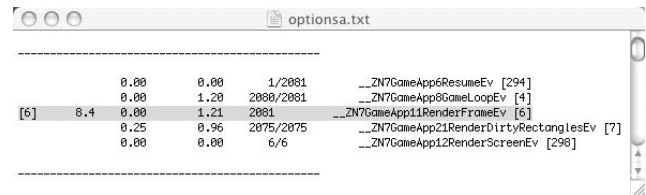
If you examine the flat profile, you will most likely see entries for functions you didn't write or call. When you write programs using Apple's frameworks, the framework functions you call in turn call lower-level functions. These lower-level functions are the unfamiliar functions appearing in your flat profile. Including the lower-level functions in the profile may seem like an annoyance. You didn't write these functions so you can't do anything to speed them up. However, they can provide clues to performance problems in your code, which I know from personal experience. I wrote a program that called `CopyBits()` to move graphics from an offscreen buffer to the screen, both of which had 32 bits of color per pixel. I profiled the program with `gprof` and learned that my program spent the greatest amount of time in a low-level function `vecCGSConvertXRGB8888toARGB8888()`. This function converts a 32-bit color pixel that does not use the highest 8 bits to a 32-bit color pixel that uses the highest 8 bits to store an alpha value, which measures the amount of transparency in a color. Even though both of my buffers used 32-bit color, they used different 32-bit color formats, which forced `CopyBits()` to perform a conversion on each pixel, slowing my application. Without seeing the calls to `vecCGSConvertXRGB8888toARGB8888()` in the flat profile, I would have never known I had a problem.

### Call Graph Profile

The flat profile alerts you to the functions in your program where the program spends the most time, but it leaves out some information you may need to optimize your code. If your program spends a lot of time in a function you didn't write, learning this information isn't going to help much because you can't change the function to make it run faster. You want to know the areas of your code that call the slow function because you can change the code in those areas. A second situation where the flat profile comes up short occurs for all but the simplest programs. Good programmers like you divide a large task into a series of smaller functions to make your code easier to write and understand. When you divide a large task into smaller functions, the flat profile can tell you how much time your program spends in each of the smaller functions, but it can't tell you how much time the program spends on the large task as a whole.

The call graph profile provides the extra information you need to optimize your code. The call graph profile contains one listing for each function your program calls. A call graph profile listing has one line of profiling information about the function itself, one line for each of its parents (the routines that called this function), and one line for each of its children (the routines this function calls). If you have a slow function that you didn't write, looking at its parent listings in the call graph profile allows you to backtrack and find the functions in your code that caused your program to call the slow function. If you break up a large task into smaller functions, the call graph profile lets you examine the task as a whole. By going to the function that calls the smaller functions, you can see how much time your program spends on the large task. The function's child listings reveal the amount of time your program spends on each function in the task. By including a function's parents and children in its call graph profile listing, you can examine larger areas of your program than you can with the flat profile.

Figure 5.3 shows a sample call graph listing. Looking at the figure you're wondering which line is the function, which lines are the parents, and which lines are the children. Let's start with the function itself, also known as the primary listing. Figure 5.3 highlights the primary listing. The primary listing is the only listing with a number in brackets in the left column. The listings above the primary listing are the parent listings, and the listings below the primary listing are the child listings.



**Figure 5.3**  
Call graph profile listing.

The call graph report provides six pieces of information for the primary listing and four pieces for both the parent and child listings. Table 5.2 lists each column and the information it provides for the primary, parent, and child listings.

`gprof` sorts the functions in the call graph report by the amount of time your program spent in the function and its descendants, the `%time` column for a primary listing. The functions with the highest amount of time appear first and have the lowest indices.

When looking at the call graph you will see entries with one parent listing whose function name is `<spontaneous>`. If `gprof` can't determine a function's parents, it creates a spontaneous parent listing with no profiling data. Signal handlers are the most likely functions to produce spontaneous parent listings. A *signal* is an event another program, such as the operating system, sends to your program. If you try to access a null pointer in your Mac OS X program, the operating system sends your program a `SIGBUS` signal, and your program handles the signal by crashing. Because signals originate outside your program, the parent of the signal handler is a function in another program. `gprof` has no way of knowing which function in the other program called the signal handler in your program, creating the need for the spontaneous parent listing.

**Table 5.2 Call Graph Profile Columns**

Column	Primary Listing	Parent Listing	Child Listing
<code>index</code>	A number uniquely identifying the function in the <code>gprof</code> report.	Empty	Empty
<code>%time</code>	The percentage of time your program spent in this function and its descendants.	Empty	Empty
<code>self</code>	The amount of time your program spent in this function.	The amount of time your program spent in the primary listing function when this parent called it.	The amount of time your program spent in this child when the primary listing function called it.
<code>descendants</code>	The amount of time your program spent in this function's descendants when this function called them.	The amount of time your program spent in the primary listing function's descendants when this parent called the primary listing function.	The amount of time your program spent in this child's descendants when the primary listing function called this child.
<code>called</code>	The number of times your program called the function.	Contains two numbers separated by a slash. The first number is the number of times this parent called the primary listing function. The second number is the number of times your program called the primary listing function.	Contains two numbers separated by a slash. The first number is the number of times the primary listing function called this child. The second number is the number of times your program called this child.
<code>name</code>	The function's name.	The parent function's name.	The child function's name.



## Cycles of Recursion

The call graph profile contains a listing for each cycle of recursion in your program. A *cycle of recursion* occurs when a function calls another function, and the second function ends up calling the first one. The second function does not have to directly call the first one to create a cycle of recursion. Suppose function A calls function B. B calls C, and C calls D. If D calls A, functions A and B form a cycle of recursion, even though B didn't directly call A.

Cycles of recursion have the potential to become large. If functions A and B call each other and functions B and C call each other, functions A, B, and C form a cycle, even though A and C didn't call each other. If C calls D and D calls C, gprof adds D to the cycle, even if D didn't call A or B.

Why do cycles of recursion matter? They matter because they complicate the calculation of the `%time` and `descendants` columns in the call graph profile. If function A calls function B and B ends up calling A, function A becomes a descendant of itself. How much of function A's execution time belongs in the `self` column and how much belongs in the `descendants` column?

gprof solves the problem of a function becoming a descendant of itself by creating a listing in the call graph profile for each cycle of recursion in your program. The name of the cycle is `<cycle X as a whole>` where *X* is the cycle number. Cycle numbers start with 1, with one number for each cycle of recursion.

A cycle of recursion has no parents. The cycle's children are the functions in the cycle. The `self` and `descendants` columns for the cycle's functions reflect the time spent in the cycle of recursion. The `called` column reflects the number of times your program called the function when it was in the cycle.

## gprof Options

The flat profile and call graph profile reports I described in the previous section are the defaults for gprof. gprof lets you supply options to it to customize the information that gprof generates. There are nine options you can supply, and you can supply multiple options with one gprof call. The following call:

```
gprof -a -b AppTitle.app/Contents/MacOS/AppTitle > results.txt
```

Combines the `-a` and `-b` options to suppress the display of statically declared functions and suppress the description of each column in the flat profile and call graph reports.

### -a Option

The `-a` option eliminates statically declared functions from the gprof report. A statically declared function is visible only in the file that defines it. C and Objective C functions as well as C++ functions that are not members of a class are statically declared if they use the `static` keyword in the declaration.

```
static void MyFunction(void);
```

When gprof finds a statically declared function, it adds the function's profiling data to the function loaded before the static function in the executable file. The function loaded before the static function may not be a function your program called. Therefore a gprof report created with the `-a` option may contain functions that would not appear if you ran gprof without the `-a` option.

## –b Option

`gprof` provides a description of each column in the flat profile and call graph profile before displaying the reports. Running `gprof` with the `–b` option removes the descriptions from the `gprof` report, saving a little space in the report.

## –e Option

The `–e` option requires you to supply a function name. The function you supply and its descendants will not have listings in the call graph profile. The flat profile stays the same. If another function calls one of the descendants you suppressed, that descendant will have a call graph profile listing.

Supplying a function at the upper levels of a deep call stack hierarchy can suppress many functions. If you supply the `main()` function the call graph profile will be empty because every function in your program is a descendant of `main()`. If you want to suppress multiple functions from the call graph profile, you must enter `–e` before each function.

```
gprof –e FirstFunction –e SecondFunction
AppTitle.app/Contents/MacOS/AppTitle > results.txt
```

For C++ programs you should supply the function name in the following form:

```
ClassName::FunctionName
```

For Objective C programs you should supply the function name in the following form:

```
–[ClassName MethodName]
```

## –E Option

The `–E` option does everything the `–e` option does and goes one step further. It excludes the undesirable functions and their descendants from the total and percentage time computations. Suppose you profiled a function for 30 seconds and the functions you want to exclude from the call graph profile ran for 10 seconds. `gprof` calculates the percentage time of each function using 20 seconds instead of 30. If your program spent 3 seconds in a function, `gprof` would report a percentage time of 15 percent (3 out of 20) with the `–E` option and 10 percent (3 out of 30) without.

## –f Option

The `–f` option has the opposite effect of the `–e` option. Only the functions you supply and their descendants will have listings in the call graph profile. The flat profile does not change. If you supply multiple functions, you must enter `–f` before each function.

```
gprof –f FirstFunction –f SecondFunction –f ThirdFunction
AppTitle.app/Contents/MacOS/AppTitle > results.txt
```

If you're interested only in the flat profile, the `–f` option is right for you. Supply one function that has no descendants, and the call graph profile virtually disappears.

## –F Option

The `–F` option does what the `–f` option does and more. It includes only the functions you supply and their descendants when calculating total time and percentage time. Use the `–F` option when you care about the performance of only certain areas of your program. In a game the area of the program you would be interested in is the game loop that runs as the player plays. Supply your game loop function with the `–F` option, and the call graph profile will contain the functions your program calls as the player plays the game. The total time and percentage time for each function will reflect the time the player spent playing.

## –s Option

Running `gprof` with the `–s` option produces a file called `gmon.sum`, which contains an accumulation of profiles for a given application. Use the `–s` option if you want to profile a program multiple times and see the results of all the profiles combined.

When you run `gprof` with the `–s` option, `gprof` generates an empty output file so don't bother specifying one. To write the results of the accumulated profiles to a file, first run `gprof` with the `–s` option to update the `gmon.sum` file.

```
gprof –s AppTitle.app/Contents/MacOS/AppTitle
```

After updating the `gmon.sum` file, run `gprof` with `gmon.sum`. Avoid the `–s` option and supply an output file. `gprof` will write the results to the file.

```
gprof AppTitle.app/Contents/MacOS/AppTitle gmon.sum > results.txt
```

## –S Option

Running `gprof` with the `–S` option produces four order files. The order files are text files with one line for each function your program calls. Table 5.3 lists the order files and the method `gprof` uses to sort each file.

What do you do with the order files? You supply one of the four order files to the linker. Supplying an order file to the linker tells the linker to place the functions in the executable file in the order specified by the order file. Without an order file the linker organizes code by its source code file, placing functions in the same source code file near each other in the executable file. The point of using an order file with the linker is to keep the functions your program calls the most close to each other. Keeping the most frequently called functions near each other reduces the number of virtual memory pages your application uses as it runs. Reducing the number of virtual memory pages reduces the chances of the operating system having to page out your program's code from RAM to disk, improving your program's performance.

Should you decide to link an order file to your program, remember two points. First, wait until your code is ready for release before linking an order file. There's no point in tinkering with the ordering of the code in the executable file until you finish writing the code, fixing the bugs, and optimizing the code. Second, make sure you run your program the way most users would run it. You want to generate order files that reflect the way users will run your program.

**Table 5.3 gprof Order Files**

Order File	Sorting Method
<code>gmon.order</code>	Sorts the functions by keeping functions that call each other together.
<code>callf.order</code>	Sorts the functions based on the number of times your program called them. The functions your program calls the most appear first.
<code>callo.order</code>	Sorts the functions according to the order in which your program calls them. The first function your program calls appears first.
<code>time.order</code>	Sorts the functions based on the amount of CPU time your program spent in them. The most time-consuming functions appear first.

To add an order file to the linker, go to the Other Linker Flags setting in the Xcode project information panel and add the flag.

```
-sectorder __TEXT __text OrderFile
```

Where *OrderFile* is the name of the order file. Choose Linking from the Collections pop-up menu to reach the Other Linker Flags setting.

When using the `—S` option, your program must generate debugging symbols. Otherwise no information will appear in the order files. You must select the Generate Debug Symbols checkbox to generate debugging symbols. Choose Code Generation from the Collections pop-up menu to reach the Generate Debug Symbols checkbox. You must also be sure to avoid linking your program with any order files.

Creating the `gmon.order` file takes a lot of time, wasted time if you plan on using one of the other three order files. To avoid creating the `gmon.order` file, pass the `—x` option along with the `—S` option.

```
gprof —S —x AppTitle.app/Contents/MacOS/AppTitle
```

## —z Option

A `gprof` profile normally contains only the functions your program called when it ran. With the `—z` option the profile contains a listing for every function in your program and every function in the frameworks you link to your program. Use the `—z` option to determine the functions your program never called.

The output file `gprof` creates using the `—z` option can be large; an application I wrote that generated a 368 KB output file with no options generated a 30.7 MB output file with the `—z` option. If you use Apple’s frameworks in your code, you’re going to get large output files. Carbon and Cocoa are enormous, each containing thousands of functions. There’s no way your application is calling all of them, leaving thousands of uncalled functions to appear in a `gprof` report with the `—z` option.

# Chapter 6

## CHUD Tools

Sampler and `gprof` can help you find the slow spots in your code. But to learn more about your program's performance, use the Computer Hardware Understanding Developer (CHUD) Tools. The CHUD Tools contain both graphical and command-line tools to measure your program's performance and provide tips to make your code run faster. This chapter covers the most important CHUD Tools: Shark, Saturn, `amber`, `simg4`, `simg5`, and `acid`.

### Shark

If you're going to learn to use only one performance tool, you should learn Shark. Shark is a source line profiler, telling you the amount of time your program spends in each function and the amount of time your program spends in each line of code. Some additional things Shark can do include.

- Recording performance events like cache hits, cache misses, virtual memory page-ins, memory reads, and memory writes.
- Providing advice to fix performance problems in your code.
- Breaking your code down to its assembly language statements.
- Profiling Java programs.

In version 4.0 Shark began to incorporate many of Sampler's capabilities, including the ability to profile selected functions and the ability to sample during memory allocations.

Like Sampler, Shark periodically takes a sample and records the functions on the call stack. Additionally, Shark keeps track of the amount of time your program spends in each function and can keep track of performance events.

Shark is a powerful program with many options, which can make learning Shark and reading this chapter difficult. As you read this chapter, remember that you don't have to use every option Shark offers. You can start by using Shark as a normal profiler to find where your program spends the most time. As you grow more comfortable with Shark, you can explore the options and harness the power the options provide.

### Configuring Shark

If all you care about measuring is where your program spends its time, running Shark is easy. Choose a program to sample and click the Start button to start profiling. But if you configure Shark before profiling you can measure a lot more. If you don't want to be overwhelmed with configuration options right now, you can skip ahead to the section "Profiling Your Program" and start using Shark. Java programmers should read the "Java Traces" section before skipping ahead.

Choose `Config > Edit` to open Shark's configuration window. Shark comes with over a dozen profiles, which I detail in upcoming sections. You can create customized profiles by selecting the profile closest to your needs and clicking the Duplicate button. Double-click the copy to change its name.

### A Brief PowerPC Processor History

Because the CHUD Tools report performance problems on specific PowerPC processors, you should know a little bit about the processors. The earliest Macs capable of running Mac OS X use the PowerPC 750 processor, also known as the G3. Macs with a G3 processor include the early iMacs without a flat-panel monitor, early iBooks, G3 Powerbooks, and blue and white towers. If your program runs well on the PowerPC 750, it will run well on any Mac that can run Mac OS X.

In 1999 Apple started shipping G4 Macs. The first G4 Macs used the PowerPC 7400 chip. The biggest improvement in the 7400 from the 750 is the AltiVec unit, also known as the Velocity Engine by Apple and the vector unit by IBM. The AltiVec unit uses 128-bit registers that can store four 32-bit values or eight 16-bit values. AltiVec instructions work on these 128-bit registers, which means a single instruction works on multiple data (SIMD). By having one instruction for all the data in the register instead of one instruction for each piece of data, AltiVec code can run four to eight times faster.

Shortly after the first G4 Macs appeared, the PowerPC 7410 replaced the 7400. The 7410 is a more energy efficient version of the 7400. To keep up with Intel and AMD's processors, Motorola developed the PowerPC 7450 chip, which is an improved version of the 7410. What distinguishes the 7450 from other PowerPC chips is Level 3 cache, which stores recently used instructions and data. The 7450 began to appear in high-end G4 towers in 2001 and started to appear in all G4 towers in September 2002. G4 Macs can have one of three processor architectures: the PowerPC 7400, 7410, and 7450.

In 2003 Apple started shipping G5 Macs, which use the PowerPC 970 processor. IBM developed the PowerPC 970FX, which is a more energy efficient version of the 970, in 2004. All Macs with a G5 processor, including the pizza box iMac, use either the PowerPC 970 or 970FX processor. The PowerPC 970 is the first 64-bit processor to appear in Macs. The G3 and G4 Macs have 32-bit processors, which limit the memory address space to 4 GB. 64-bit processors have a theoretical limit of  $(2^{64})$  bytes of memory, which means G5 Macs can have more than 4GB of memory installed on them. The 970 also runs at higher clock speeds, has a faster system bus, has faster memory, and can execute more instructions simultaneously.

### Time Profiles

As its name suggests, the time profile measures the amount of time your program spends in each function. Only C, C++, and Objective C programs can use the time profile. Shark has its own profiles for Java programs.

The all thread states time profile works the same way as watching for timed samples in Sampler, which I covered in Chapter 4. Shark takes a sample at a time interval that you specify. The advantage of using the all thread states time profile is simplicity. All you have to tell Shark is the sampling interval.

The all thread states time profile's simplicity doesn't mean the regular time profile is difficult to use. There are more settings you can customize in the regular time profile, giving you greater flexibility at the cost of slightly higher complexity. The regular time profile has the following advantages:

- Higher sampling rates, up to 20000 samples per second. The all thread states time profile has a limit of 1000 samples per second.
- The ability to record low-level performance events like cache misses.
- The ability to set a limit on how long Shark profiles your program.
- The ability to profile more than one program at a time.

## System Trace

Use the system trace profile to learn how your application interacts with other programs. The system trace profile examines every running program. It records every system call and every virtual memory fault. Shark also records a sample for every scheduling interval. A scheduling interval is a slice of time the operating system gives to a thread. Scheduling intervals let you run multiple programs at one time. Applications you're using get more scheduling intervals than background programs.

## Function Trace

In the function trace profile, Shark records a sample when your program calls one of the functions you tell Shark to look for. The function trace profile is the equivalent of watching for function calls in Sampler, which I covered in Chapter 4. To add a function to the watch list, type the name of the function in the text box next to the Add button and click the Add button.

## Java Traces

The Java traces profile Java programs. There are three built-in Java traces. Java alloc trace records a sample when the program allocates memory. Java method trace records the entry into each method your Java program calls. Java time trace records a sample at regular time intervals.

To profile a Java program in Shark, the program must have the virtual machine option `-XrunShark` set. To set the virtual machine option in Xcode:

- 1) Double-click the target name in the Groups and Files list to open the target settings window.
- 2) In the Info.plist Entries section of the target settings window, select Pure Java Specific.
- 3) Add the flag `-XrunShark` in the Additional VM Options text field.

Shark can profile only running Java programs. If you try to launch a Java program from Shark, Shark will launch the program, but it won't do any profiling. To save yourself some aggravation, make sure your Java program is running before profiling it with Shark. There are two ways to ensure your program is running.

- Launch your program before you start profiling with Shark.
- Launch your Java program from Xcode by choosing **Debug > Launch Using Performance Tool > Shark**. When you launch from Xcode, your program launches before Shark starts profiling.

## Data Cache Miss Profiles

Shark comes with five data cache miss profiles, which tell Shark to take a sample when there's a data cache miss. A data cache miss occurs when the CPU is unable to find the data it needs in the cache. Shark's profiles measure data cache misses in the Level 2 cache.

The G4 profile measures data cache misses on PowerPC 7400 and 7410 processors. The G4+ profile measures data cache misses on PowerPC 7450 processors. The G5 profile measures data cache misses on PowerPC 970 and 970FX processors. The Pentium 4 profile measures cache misses on Pentium 4 processors. The Pentium M profile measures cache misses on Pentium M processors. You must be using an Intel Mac to use the Pentium 4 and Pentium M profiles.

If you have an Intel Mac, you may be wondering which Pentium profile to use to measure cache misses on Intel Macs. Use the Pentium M profile. The Core Duo and Core Solo processors that power Intel Macs inherit from the Pentium M architecture.

### **Malloc Trace Profile**

When you use the malloc trace profile, Shark records a sample every time your program makes a memory allocation. The malloc trace profile does what watching for memory allocations does in Sampler, which I covered in Chapter 4.

### **Memory Bus Bandwidth Profiles**

When you use a memory bus bandwidth profile, Shark records every memory read and write in your program. Shark slices the reads and writes into individual beats. A beat is a transfer the size of the data bus, which is 8 bytes for G4 Macs and 16 bytes for G5 Macs. If your program reads 128 bytes of memory, there will be 8 or 16 beats, depending on the data bus size. Shark records each beat. Recording beats measures the memory bus bandwidth.

There are two memory bus bandwidth profiles: one for the U2 memory controller and one for the U3 memory controller. G4 Macs should use the U2 profile, and G5 Macs should use the U3 profile.

### **Static Analysis Profile**

The static analysis profile differs from the rest of the Shark profiles in that it doesn't require your program to be running. Shark examines your code and reports its analysis to you. There are two types of static analysis: function browser and problem search.

The function browser analysis creates one listing for each function in the program or file you're profiling. By using the function browser analysis, Shark can examine your source code and provide code tuning advice to make your code run faster.

The problem search analysis examines your code for possible performance problems. You specify the CPU to use to look for problems and specify the severity of problems to look for. The problem search analysis became more valuable with the introduction of the Intel Macs. It is the only Shark profile you can run on a PowerPC Mac to look for performance problems on Intel Macs and vice versa. Use the problem search analysis to look for performance problems on a variety of CPUs without having to run your program.

### **VM Faults Profile**

The VM Faults profile tells Shark to take a sample when virtual memory page faults take place. A page fault occurs when the operating system can't find a page of memory in physical memory. You get to specify how many page faults occur between samples.



## Windowed Time Facility

If you're running Shark 4.4 or later, you will see two profiles called Time Profile (WTF) and System Trace (WTF). You might be thinking "WTF does WTF stand for?" WTF stands for windowed time facility. Instead of storing every sample, profiles that use the windowed time facility store only the most recent samples. You get to specify how many samples to store in the recent sample history.

Why would you want to store only the most recent samples? Storing the most recent samples lets you profile for a long time. The Shark profiles that store every sample are designed to profile your program for a short time, no longer than a couple of minutes. If your program starts to run slowly after running for 20 minutes, you're out of luck because Shark stopped sampling before the slowdown occurred. By using the windowed time facility, you can keep your program running until your program starts to run slowly. When your program starts to run slowly, you can pause Shark, look at the recent samples, and see what caused the slowdown.

## Setting the Sampling Rate

When you select a configuration from the configuration list, the plug-ins list shows the Shark plug-ins the configuration uses. If the configuration you select uses the CHUDDDataSource plug-in, you have a lot of control over the sampling rate. Click the Sampling tab to set the sampling interval. The sampling interval can be either a period of time (specified in microseconds, milliseconds, or seconds), a number of CPU events, or a number of operating system events. The time profile uses a period of time as the sampling rate. The data cache miss profiles use a number of CPU events, and the VM faults profile uses a number of operating system events.

## Recording Performance Events

What sets Shark apart from other performance tools is its ability to record performance events. There are three categories of performance events.

- CPU events, such as cache hits, CPU stalls, and wrongly predicted branches.
- Memory events, like reads and writes. The reads and writes can occur in RAM, in PCI memory, or in AGP memory.
- Operating system events, such as file reads, file writes, system calls, and exceptions.

To set the performance events to record, your profile must use the CHUDDDataSource plug-in. If you select CHUDDDataSource from the list of available plug-ins and click the Counters tab, the lower right portion of the configuration window shows a list of available events to record and a list of events you're recording, which you can see in Figure 6.1. To add an event to the recording list, select it from the available events list and choose Counter from the menu in the Mode column.

Mode	Type	Name
None	CPU	dL1 Writes Hit Shared
None	CPU	ITLB Search Cycles > Threshold
None	CPU	dL2 Misses
None	CPU	Instr Bkpt match
None	CPU	dL1 Snoop Interventions
None	CPU	dL1 Load Hits
None	CPU	L2 Snoop Interventions

**Figure 6.1**

Performance event list.

If you choose Trigger from the menu in the Mode column, the event becomes a trigger. When you make an event a trigger, Shark records a sample when the trigger event occurs instead of when a period of time passes. Only operating system events and CPU events can be triggers. The advantage of using a performance event as a trigger is you can profile multiple programs without having to profile every running program. When you use a timer trigger, Shark profiles every program or profiles just one program. If you're interested in how much time your program spends in each function, use a timer as a trigger.

All Macs can use operating system events as triggers. Macs with the Power PC 7450 and 970 processors as well as Macs with version 1.3 or later of the PowerPC 7410 processor can use CPU events as triggers. Intel Macs can also use CPU events as triggers. If you make a performance event a trigger, click the Sampling tab to tell Shark how many trigger events must occur before Shark takes a sample.

Sometimes when you try to add a performance event, Shark won't be able to add it. There are two possible reasons for Shark being unable to add an event to the list of events to record. First, you exceeded the event limit. The exact limits depend on your computer, but every PowerPC Mac can record at least four CPU events, four memory events, and four operating system events. Intel Macs can record two of each event type. Second, the event you want to add doesn't apply to your Mac. The available event list shows the events for all Macs. Some of these events aren't available on your computer. The program PMC Index, which is part of the CHUD Tools, lets you see the performance events available for each CPU, memory controller, and version of Mac OS X.

## Advanced Configuration Options

Choosing Advanced from the View pop-up menu shows a list of all available plug-ins. Select a plug-in's checkbox to add it to the selected profile. There are three categories of plug-ins.

- Data source plug-ins generate the profiling data.
- Analysis plug-ins create intermediate analysis for viewers.
- Viewer plug-ins display the profiling data.

If you look at the built-in Shark profiles, you'll see that each profile has one data source. While you can have multiple data sources in a profile, there's not much point in doing so. Each data source profiles one specific category. If you need to use multiple data sources, you're better off creating multiple profiles.

Select a plug-in from the list to configure the plug-in. The two plug-ins you're most likely to configure are CHUDDataSource and MONsterProfileAnalysis. You cannot configure many of the plug-ins and the rest of the plug-ins' advanced configuration options are the same as the simple configuration options.

## Choosing a Trigger

Select the CHUDDataSource plug-in and click the Sampling tab to choose the trigger Shark uses to record a sample. Because many of the trigger settings in the simple and advanced views are identical, I'm going to focus on the trigger settings that have no equivalent in the simple configuration. For the identical settings, it doesn't matter where you set them. They are usually easier to set from the simple view.

The simple view gives you three possible triggers: timer, CPU event, and operating system event. The advanced view gives you a fourth trigger: the function `chudRecordUserSample()`, which is part of the CHUD framework. This function tells Shark to take a sample, allowing you to control Shark from your source code. To use `chudRecordUserSample()` with your program, you must add the CHUD framework to your project and add calls to `chudRecordUserSample()` in your program.

When you use a timer trigger, the timer fires at the exact time you specify. The Fuzz checkbox lets you randomize the sampling rate. You can randomize the sampling rate up to 50%. With a fuzz percentage of ten percent and a sampling rate of one millisecond, Shark takes a sample after 0.9 to 1.1 milliseconds pass.

Why would you want to randomize the sampling rate? The sampling rate can affect how your program executes when Shark profiles it. Randomizing the sampling rate reduces its impact, allowing Shark to profile your program more accurately.

The Sample Limit checkbox and text field lets you specify how many samples Shark takes before stopping. You can set both a time limit and a sample limit. Shark stops when it reaches one of the limits.

## Choosing the CPU and Memory Controller

Select the CHUDDDataSource plug-in to choose the CPU and memory controller Shark uses for profiling. Shark uses your Mac's CPU and memory controller initially, but you can tell Shark to pretend your Mac has a different CPU or memory controller. Pop-up menus at the bottom of the window let you choose the CPU and memory controller. By choosing a different CPU or memory controller you can simulate how your program runs on another Mac, alerting you to performance problems on other Macs.

If you tell Shark to use a different CPU or memory controller, don't count performance events. Shark can count performance events only for your Mac's CPU and memory controller.

Shark 4.2 added support for Intel processors. If you're running Shark 4.2 on a PowerPC Mac, you would like to use Shark to see how your program performs on Intel processors. Unfortunately, you can't tell Shark to pretend your PowerPC Mac has an Intel processor. You must profile your program on an Intel processor to see how the program performs.

## Filtering Programs to Profile

After choosing the CPU and memory controller to use, you'll notice the configuration window has tabs with the names of the chosen CPU and memory controller. Click the CPU tab to filter the programs Shark profiles. The Privilege Level radio button group tells Shark the privilege level of programs to profile. There are two privilege levels: user and supervisor. Applications have user privilege. Programs with supervisor privilege are low-level programs the operating system uses, such as kernel extensions. Profiling programs with user privilege is sufficient in most cases.

The Process radio button group tells Shark to profile marked or unmarked processes. In most cases you want to profile marked processes, with the marked processes being the programs you want to profile. To mark a process, open the process marker panel by choosing Sampling > Process Marker. The process marker panel contains every program you have running. Some processes cannot be marked. These processes have gray text. To mark a process, select it. The background color in the panel changes from white to blue to signify that the program has been marked. To unmark a marked process, select it from the panel.

Should you change any settings in the Privilege Level and Process radio button groups, they have meaning only if you tell Shark to sample all running programs. If you tell Shark to profile only one program, marking processes and telling Shark to sample only programs with user privilege have no effect. Shark profiles only the program you tell it to profile.

## Setting Events to Record

Shark uses the PowerPC and Intel Core's performance monitor counters (PMC) to record performance events. On PowerPC Macs there are 4–8 PMCs for CPU events, 4–6 PMCs for memory events, and 4 PMCs for operating system events. Intel Macs have 2 PMCs for CPU, memory, and operating system events. When you use the simple configuration view to record performance events, Shark assigns a PMC for each event for you. The advanced configuration view for the CHUDDataSource plug-in lets you manually set the event each PMC records.

The configuration window has a CPU model tab for recording CPU events, a memory controller tab for recording memory events, and a Mac OS tab for recording operating system events. Click the appropriate tab to configure the PMCs for a group of performance events. Figure 6.2 shows the PMCs for CPU events. The PMCs for the other types of performance events look similar. Choose an event to record from the combo box. Click the button next to the PMC to turn on the PMC. The button's title changes to Counter when you turn on a PMC. Click the button a second time to make the event a trigger.

## G5 Specific Events

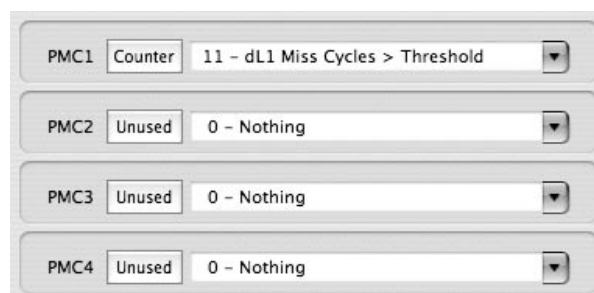
The G5 processor has its own set of CPU events. If you have a G5 processor on your Mac, you can configure these events.

## Event Multiplexing

G3 and G4 processors have only direct events, where the performance monitor counter (PMC) directly records one event. IBM introduced indirect events with the G5 processor. With indirect events the CPU multiplexes some of the performance events. These multiplexers give the G5 a greater number of possible events to record. Shark lets you configure the multiplexers.

The G5's multiplexers consist of four general lanes and one spec lane. For each general lane you can specify the multiplexer to use as well as the execution unit from which the multiplexer records events. Table 6.1 lists the multiplexers for the general lane. The spec lane can use any of four speculative event multiplexers. The spec lane is for speculative event counting, which occurs when an instruction group stalls or the global completion table (GCT) becomes empty.

By setting up the lanes, you determine the events each of the G5's eight PMCs can record. Lanes 0 and 2 affect PMCs 1, 2, 5, and 6. Lanes 1 and 3 affect PMCs 3, 4, 7, and 8. The spec lane affects PMCs 5 and 7. Suppose you tell lane 0 to use TTM0, which you set to record events from the FPU. By setting lane 0 to use the FPU, only PMCs 1, 2, 5, and 6 can record all the events that occur in the floating-point units.



**Figure 6.2**

Setting performance monitor counters.

There are pop-up menus to select the multiplexer to use for each lane as well as menus to set the execution unit for each general-purpose multiplexer, as you can see in Figure 6.3. Keep three things in mind. First, the general lanes can record events from the load/store units instead of using the multiplexers listed in Table 6.1. Second, choose GPS from the TTM1 pop-up menu if you want TTM1 to record events from the storage subsystem. Third, the TTM3 pop-up menu lets you specify whether bytes 2 and 3 come from the lower or upper 32 bits of LSU1. The pop-up menu has the following entries:

- LSU1 2|3, which means bytes 2 and 3 come from the lower 32 bits.
- LSU1 2|7, which means byte 2 comes from the lower 32 bits and byte 3 comes from the upper 32 bits.
- LSU1 6|3, which means byte 2 comes from the upper 32 bits and byte 3 comes from the lower 32 bits.
- LSU1 6|7, which means bytes 2 and 3 come from the upper 32 bits.

### CPU Instruction Matching

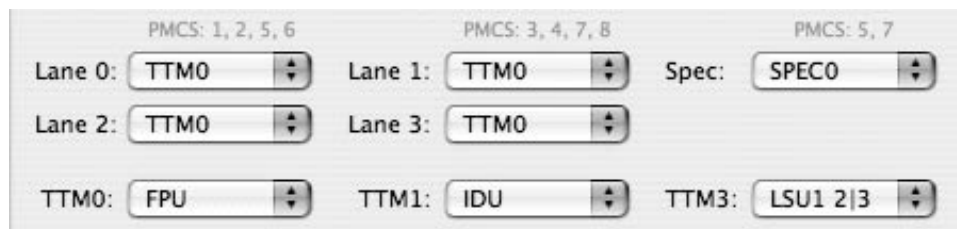
The G5 processor has an instruction matching facility you can configure to limit the assembly language instructions Shark samples. There are two decisions to make. First, you must decide whether to use the instruction fetch unit (IFU) or instruction dispatch unit (IDU) for instruction matching. Use the IFU when you know the instructions you want Shark to sample. Use the IDU when you care about the internal instructions, iops, generated from an assembly language instruction.

Second, you must decide which iops Shark samples using the IOP Marking radio button group. You can tell Shark to sample the following iops:

- All iops.
- Only iops expanded from architected instructions.
- Only one iop for each assembly language instruction.
- The first iop to travel to the load/store unit for each load/store instruction.

**Table 6.1 General-Purpose Multiplexers**

Multiplexer	Lanes	Execution Units It Can Record From
TTM0	0, 1, 2, 3	Floating Point Unit (FPU), Vector Permute Unit (VMX), Instruction Fetch Unit (IFU), Instruction Sequencer Unit (ISU)
TTM1	0, 1, 2, 3	Instruction Dispatch Unit (IDU), Instruction Sequencer Unit (ISU), Storage Subsystem (STS)
TTM3	2, 3	Load/Store Unit 1 (LSU1)



**Figure 6.3**

Configuring the G5 event multiplexers.

## IFU Instruction Matching

The IFU instruction matching facility, shown in Figure 6.4, lets you specify the instructions you want Shark to sample. If you were interested only in AltiVec instructions, you would tell the IFU instruction matching facility to match AltiVec instructions.

When configuring the IFU instruction matching facility, the first task is to choose an Instruction Match CAM register (IMC) row from the pop-up menu. There are six rows: rows 0 through 5. You can configure each row to match a different set of instructions. Select the Enabled checkbox to start matching instructions for a row.

To match instructions for an IMC row, you must set bit masks to determine the instructions that match for that row. There are six bits for the major opcode, 11 bits for the minor opcode, and four bits for the machine state register (MSR). The major opcode identifies the type of instruction. AltiVec instructions have a major opcode of 4, and floating-point instructions have major opcodes 59 and 63. The minor opcode uniquely identifies an instruction from others of the same type.

There are four bits you can set for the MSR.

- TA. There is no mention of the TA bit in the PowerPC documentation on the MSR so I have no idea what the TA bit does.
- PR, which is the privilege level bit. Setting PR to 1 means the CPU can handle only non-privileged instructions. Setting PR to 0 means the CPU can handle any instruction.
- FP, which is the floating-point available bit. Setting FP to 1 means the CPU can handle floating-point instructions.
- VMX, which is the vector unit available bit. Setting VMX to 1 means the CPU can handle vector (AltiVec) instructions.

You can set a bit to have any of the following values:

- Never Match. If any of the bits in the major opcode or minor opcode are set to Never Match, no instructions match.
- Match 0, which means an instruction matches if one of its opcode bits has a 0 where you set the bit to be 0.
- Match 1, which means an instruction matches if one of its opcode bits has a 1 in that particular position.
- Don't Care, which means the instruction matches automatically. Setting all the bits to Don't Care means every instruction matches, which defeats the purpose of instruction matching.

When you set more bits to Match 0 or Match 1, fewer instructions match. Setting more bits to Don't Care increases the number of instructions that match.

The screenshot shows the IFU instruction matching configuration window. It includes sections for Major Opcode, Minor Opcode, and MSR (TA, PR, FP, VMX). There are checkboxes for 'IMC Row 0', 'Enabled', 'Match None', and 'Match Any'. A table at the bottom lists instructions with their major and minor opcodes. The status bar indicates '0 of 423 PPC instructions selected' and 'All Rows' and 'All Instructions' are checked.

Mnemonic	Major	Minor	Match Rows
add	011111 (31)	00100001010 (266)	
addc	011111 (31)	00000001010 (10)	
addco	011111 (31)	01000001010 (522)	
adde	011111 (31)	00010001010 (138)	
addeo	011111 (31)	01010001010 (650)	
addi	001110 (14)	00000000000 (0)	
addic	001100 (12)	00000000000 (0)	
addic.	001101 (13)	00000000000 (0)	
addic.	001111 (15)	00000000000 (0)	

**Figure 6.4**

IFU instruction matching.

## IDU Instruction Matching

The IDU instruction matching facility, shown in Figure 6.5, lets you specify the instructions for Shark to sample based on the iops the instructions generate. The IDU uses four predecode bits for instruction matching.

- Branch bit (B), which creates an iop from a branch instruction when the bit is set.
- Split bit (S), which splits an instruction into multiple iops when the bit is set.
- First bit (F), which makes the iop the first instruction in a dispatch group when the bit is set.
- Last bit (L), which makes the iop the last instruction in a dispatch group when the bit is set.

Initially the IMRMASK and IMRMATCH are 0000, which means all instructions pass. Setting a bit in the IMRMASK blocks the instructions where you set the bit; setting the B bit blocks all instructions where the B value is 1, BSFL values 1000 through 1111. What do you do if you want to set the instructions you want to pass instead of setting the ones you want to block? That's where the IMRMATCH comes in. Set the appropriate bits in the IMRMASK and set the same bits in the IMRMATCH. Setting a bit to 1 in the IMRMATCH that doesn't have the corresponding bit set in the IMRMASK blocks all instructions.

## Setting Intel Performance Events

Shark has three architecture options for Intel Macs: Pentium 4, Intel Core, and Intel Core 2. You're going to want to use either the Intel Core or Intel Core 2 architectures. 32-bit Intel Macs have the Intel Core architecture, and 64-bit Intel Macs have the Intel Core 2 architecture.

The downside of Intel Core and Intel Core 2 for Shark is that these architectures have only two performance monitor counters (PMC). Having two PMCs means you can count only two CPU, memory, and operating system events.

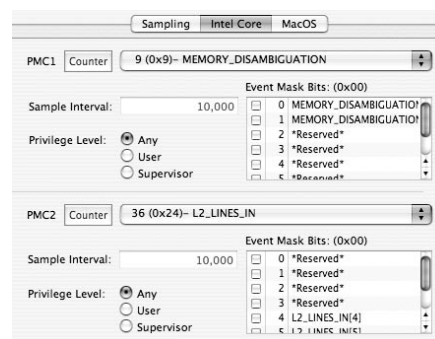
Figure 6.6 shows the user interface to configure an Intel Core PMC. Choose an event from the pop-up menu. When you choose an event, Shark fills the Event Mask Bits list with the mask bits you can set for that event. Select the checkbox next to a mask bit to set that bit.

After telling Shark what a PMC should record, you must turn on the PMC. Click the Unused button. The button's text should change to Counter. Click the button a second time to make the PMC a trigger. When you make a PMC a trigger Shark records a sample when one of the events the PMC is recording occurs in your program.



**Figure 6.5**

IDU instruction matching.



**Figure 6.6**

Intel performance event counting.

## Creating Profiling Equations

The MONsterProfileAnalysis plug-in lets you create profiling equations. Profiling equations take the performance events Shark counts and transform them into useful data. The memory bandwidth profiles use profiling equations to measure the memory bandwidth in megabytes per second. The number of beats Shark recorded doesn't measure the bandwidth, but converting the beats into megabytes per second does.

The MONsterProfileAnalysis plug-in is strictly for performance event counting. Before you create profiling equations you must configure the PMCs. If you select the MONsterProfileAnalysis plug-in without configuring the PMCs, you'll get the spinning rainbow cursor, forcing you to restart Shark.

Figure 6.7 shows the user interface to create a profiling equation. At the top is a list of PMCs with a description of the event each PMC is counting. The Symbol column contains a tag for the event. Use the tag as a variable in your profiling equations.

At the bottom is a list of equations. Click the Add button to add an equation. Name the equation in the Shortcut Name column. Enter the equation in the Shortcut Equation column. Use the performance events' symbols as variables in the equations. You can also use the results of previous equations as variables. The first equation has the variable name eq01, the second equation has the name eq02, and so on.

## Configuring the Profiling Session

Choose Config > Show Mini Config Editor to attach the mini configuration editor to the Shark main window. Use the mini configuration editor to perform basic configurations without having to open the configuration window. What you can configure in the mini configuration editor depends on the Shark profile you're using. Many of Shark's built-in profiles let you specify the following information in the mini configuration editor:

- How long Shark should wait before starting to sample your program. You can specify the time to wait in microseconds, milliseconds, or seconds.
- How long Shark should profile your program before stopping. You can specify the time to profile in microseconds, milliseconds, or seconds.
- The number of samples Shark should take before stopping.

PMC	Mode	Symbol	Performance Counter Description
M/C PMC-4	counter	m1c4	Burst Write Reqs [Bus]
M/C PMC-3	counter	m1c3	Burst Read Reqs [Bus]
M/C PMC-2	counter	m1c2	Single Beat Write Reqs [Bus]
M/C PMC-1	counter	m1c1	Single Beat Read Reqs [Bus]
Shortcut Name		Shortcut Equation	
Burst Writes MB/s		((m1c4*32)*100)/1000000	
Burst Read MB/s		((m1c3*32)*100)/1000000	
Single Writes MB/s		((m1c2*32)*100)/1000000	
Single Reads MB/s		((m1c1*32)*100)/1000000	
Total Writes MB/s		eq01+eq03	
Total Reads MB/s		eq02+eq04	

**Figure 6.7**

Profiling equations.



## Profiling Your Program

When you launch Shark, the main window, shown in Figure 6.8, opens. The main window is where you tell Shark what to profile and how to profile. This window can have two or three pop-up menus. The leftmost pop-up menu contains the list of available profiles. Next to the profile list is a second pop-up menu that tells Shark what to do: analyze a file, sample a program (Process) or sample all running programs (Everything). The system trace profile and profiles that use the CHUDDDataSource plug-in can sample all running programs. Only the static analysis profile can analyze a file. All the built-in profiles can sample a process, or analyze in the case of a static analysis profile. If you create a custom profile, what you can profile depends on the built-in profile you used as a base.

If you choose to profile a single program, a third pop-up menu appears, containing all the currently running programs. Choose the program you want to profile from the menu. If the program you want to sample isn't running, choose Launch. Click the Start button to start profiling. If you launch your program from Shark, a dialog box opens asking for the program to sample. After clicking OK, Shark starts sampling.

If you choose to analyze a file, the third pop-up menu contains the list of files you can analyze. If there are no files to analyze, choose Open. Click the Start button. If you chose Open, a dialog box opens asking for the file to analyze. After clicking OK, Shark starts analyzing the file.

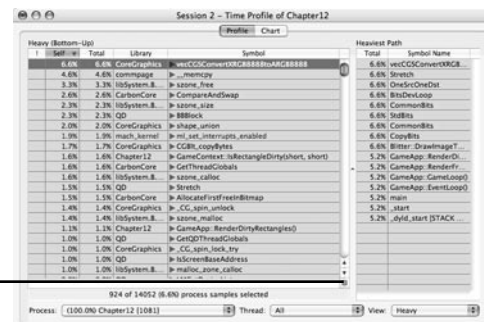
After clicking the Start button, Shark goes to work sampling programs. The sampling continues until you click the Stop button or Shark takes the configured number of samples. When you stop sampling, Shark takes some time to show the results. The reason for the delay is that Shark doesn't do its calculations and analysis until you stop sampling. Waiting to perform the calculations keeps your program running smoothly, but causes a delay when you're finished. To avoid the delay, use batch mode. Batch mode tells Shark not to show the results until you turn off batch mode. Choose Sampling > Batch Mode to turn batch mode on and off.

## Viewing Shark's Results

After Shark finishes processing the recorded samples, it opens a results window and places the results there. Figure 6.9 shows an example of the results window. Initially Shark shows the results of the dominant process, the program where the CPU spent the most time, and the dominant thread in that process. But you can use the pop-up menus at the bottom of the window to choose another program or thread. If you profile only one program, there's going to be only one program to choose in the pop-up menu.



Call Stack Button



**Figure 6.8**

Shark launch window.

**Figure 6.9**

Shark results window.

Initially the Shark results window shows the profile view, which is a table of profiling statistics. For each function your program calls, the profile view has five columns of information.

- Tuning advice column, designated by an exclamation point. If there's an exclamation point in the column, clicking the exclamation point brings up advice to make your code run faster.
- Self, which initially is the percentage of samples where this function is at the top of the call stack. For a time profile, the Self column tells you the percentage of time your program spent in this function.
- Total, which initially is the percentage of samples where this function appears in the call stack. For a time profile, the Total column tells you the percentage of time your program spent in this function and its descendants.
- Library, which is the library where the function resides.
- Symbol, which is the name of the function. To see symbols in your program, you must compile your program to generate debugging information. Refer to the section “Before You Debug” in Chapter 2 for instructions on generating debugging information.

In addition to these five columns, the profile view has one column for each performance event you told Shark to record. Shark stores the total number of times a performance event occurred. The event's corresponding column tells you what percentage occurred in a particular function and its descendants. If you don't see performance event columns, choose File > Show Advanced Settings. In the Performance Count Data Mining section, you'll see the performance events Shark counted. Select the left checkbox (the one under the eye column) for an event to tell Shark to show that event.

## Heavy and Tree Views

The initial view in the results window is the heavy view. The heavy view is the best view for finding the slow areas of your code. It shows you the calling paths where your program spends a lot of time. There's one listing in the heavy view for every function that appeared at the top of the call stack at least once when Shark took a sample. A function appears at the top of the call stack when your program is inside that function. Shark sorts the listings so the listings that consume the most time appear first. A heavy view listing for a function contains the path from the function back to the top of the call tree, which is the first function your program called. Click the disclosure triangles to move up the call tree.

When you look at a heavy view listing, remember that it represents a path from one function to the top of the call tree. A single function can appear in multiple heavy view listings. The statistics for a function in the heavy view listing reflect the samples taken in the path, not the total number of samples where the function appears. When a function shows a percentage of 20 in the Total column in the heavy view, it means your program spent 20 percent of the time in this path. The heavy view focuses on the statistics for a calling path, not the statistics for an individual function.

When you want to focus on statistics for individual functions, switch to the tree view using the View pop-up menu at the bottom of the results window. The tree view starts displaying functions at the top of the call tree. Clicking the disclosure triangles moves you down the call tree. The tree view gives you a global view of your program's performance. When a function shows a percentage of 20 in the Total column in the tree view, you know your program spent 20 percent of the time in that function and its descendants.

## Showing the Call Stack Table

The Shark results window has a button below the vertical scroll bar. Clicking the button makes the call stack table appear in the right half the results window. Selecting a row from the results window fills the call stack table with the function's call stack, the series of functions that called this function. The call stack table lets you view the call stack without having to click a series of disclosure triangles.

## Viewing Source Code

Double-clicking a function name in either the results window or the call stack table shows you the source code for that function in the Shark results window, turning the results window into a code browser. Shark highlights the line of code with the most samples.

The code browser lets you look at a function's source code in the high-level language you used to write the code or in assembly language. Looking at a function in a high-level language lets you find the lines of code where your program spends the most time. Looking at the assembly language code provides additional information, such as the number of clock cycles your program spends in each assembly language instruction.

There are three buttons at the top of the code browser that determine what appears in the code browser. Clicking the Source button tells Shark to look at the code in the high-level language it was written in. Clicking the Assembly button tells Shark to look at the assembly language code. Clicking the Both button shows both the high-level language and the assembly language code in the code browser. If a function does not have high-level source code available to view, such as functions in Apple's frameworks, Shark disables the Source and Both buttons.

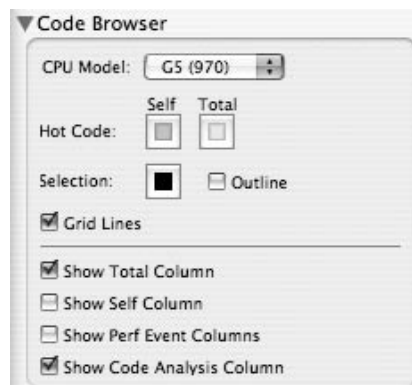
## Customizing the Code Browser

Choose File > Show Advanced Settings to open the advanced settings drawer. The advanced settings drawer lets you customize what appears in the results window and how the data appears. In the Code Browser section of the advanced setting drawer, which you can see in Figure 6.10, there are checkboxes that let you determine the columns that appear in the code browser. If you told Shark to record performance events, selecting the Show Perf Event Columns checkbox tells Shark to add a column in the code browser for each performance event Shark recorded.

The most interesting code browser setting on PowerPC Macs is the CPU Model pop-up menu. With the CPU Model pop-up menu, you can look for code problems on PowerPC 7400, 7450, and 970 processors. The CPU model affects only the assembly language code statistics.

## Viewing Assembly Language Code

The advantage of viewing assembly language code is you can see assembly language source code for every function in your program, including the functions in Apple's frameworks. If you want to see what's happening in your program when you make Cocoa function calls, look at the assembly language. Shark initially reports the following data for each assembly language instruction:



**Figure 6.10**

Shark's code browser settings.

- Total, which is the percentage of time your program spent in this instruction when the program was in this function.
- Address, which is the memory address for the instruction.
- Code, which is the instruction.
- Cycles, which is the number of processor (clock) cycles spent in this instruction. The number of processor cycles per second equals one billion multiplied by the gigahertz of the processor. A 2 GHz Power Mac G5 has two billion clock cycles per second.
- Tuning advice column, designated by an exclamation point, which provides advice to speed up your code.
- Comment, which alerts you to any performance problems, such as stalls. Deselecting the Show Code Analysis Column checkbox in the advanced settings drawer eliminates the Comment column.
- Source, which lists the file name and line number. Code you didn't write has a blank Source column.

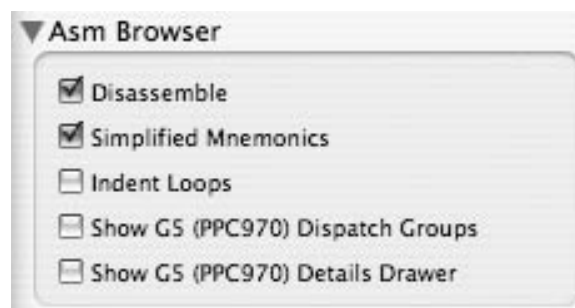
The Cycles column displays two numbers separated by a colon. The first number represents latency, and the second represents throughput. *Latency* measures the number of cycles that elapse between an instruction starting execution and the instruction producing a final result. *Throughput* measures the number of cycles between finishing instructions of this type when the corresponding pipeline is full. If an integer instruction finishes, throughput measures the number of cycles it takes for a second integer instruction to finish when the integer instruction pipeline is full. If the instruction has an asterisk in the Cycles column, the instruction causes the pipeline to be serialized, which means the pipeline must wait until the serialized instruction finishes.

Use the checkboxes in the advanced settings drawer to show more columns of information in the code browser.

### G5 Assembly Language Options

If you're examining your code using the G5 CPU model, Shark provides additional options. Selecting the Show G5 Dispatch Groups checkbox in the Asm Browser section of the advanced settings drawer, shown in Figure 6.11, adds grid lines around each group of instructions. The PowerPC 970 takes up to four non-branching instructions and one branch instruction and places them into a group. The processor takes the group of instructions, splits the group into individual instructions for execution, and regroups the instructions. By using groups of instructions, the PowerPC 970 can have more instructions in the pipeline at once, over 200 instructions.

Because a group can have up to five instructions, showing the dispatch groups can alert you to performance problems on G5 processors. If you have lots of groups with only one or two instructions in them, you're not running at peak efficiency. Groups containing four or five instructions show that your code is running efficiently.



**Figure 6.11**

Shark's assembly language browser.

Selecting the Show G5 Details Drawer checkbox in the Asm Browser section opens the details drawer, which you can see in Figure 6.12, at the bottom of the code browser. The details drawer tells you the execution unit and group dispatch slot an instruction uses. Selecting instructions in the code browser shows the following information about the instructions in the details drawer:

- The execution units these instructions use.
- The number of groups these instructions belong to.
- The average size of each group.
- The dispatch slots the instructions use.

## Assembly Language Reference

Assembly language instructions use mnemonics, making it difficult to figure out what the instruction does. Do you know what the instruction `stswi` does? There are hundreds of PowerPC assembly instructions, many with names that are as difficult to understand as `stswi`, but Shark provides help.

Clicking the Asm Help button opens a window containing the PowerPC assembly language instruction reference manual. Selecting an instruction from the code browser opens the material on that instruction in the reference window. The reference manual provides one to two pages of information on each instruction, enough to give you an idea of what the instruction does.

If you're running Shark on an Intel processor, clicking the Asm Help button opens the Intel assembly language reference manual. Shark 4.2 provides both the PowerPC and Intel assembly language references so you can read the Intel assembly language reference on a PowerPC Mac and read the PowerPC assembly language reference on an Intel Mac. Choose Help > PowerPC ISA Reference to read the PowerPC reference. Choose Help > IA32 ISA Reference to read the Intel reference.

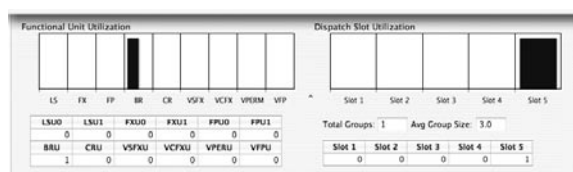
## Viewing High-Level Language Code

You can view high-level language code only for functions you wrote. For code you wrote, Shark loads the function's file in the results window. Shark initially shows the following information for each line of code:

- Total, which is the percentage of time your program spent in this line of code when the program was in this function.
- Line, which is the line number.
- Code, which is the line of code.
- Tuning advice column, which provides advice to make your code run faster.
- Comment, which alerts you to any problems in your code.

Use the checkboxes in the advanced settings drawer to show more columns of information in the code browser.

Clicking the Edit button opens the source code file in Xcode and takes you to the function you were examining in Shark. Xcode must be running for Shark to open the source code file.



**Figure 6.12**

G5 details drawer.

## Customizing What the Results Window Displays

Earlier I told you the Self and Total columns in the results window display the percentage of samples. I also told you the performance event columns display the percentage of the event. What appears in these columns depends on what you tell Shark to show in them.

The advanced settings drawer, which you open by choosing File > Show Advanced Settings, is where you customize what appears in the results window. Click the Profile tab in the results window to show the Profile Analysis section of the advanced settings drawer. Figure 6.13 shows the Profile Analysis section. The Stats Display pop-up menu tells Shark to display either a percentage or a value. If you'd prefer not to see percentages, choose Value from the menu.

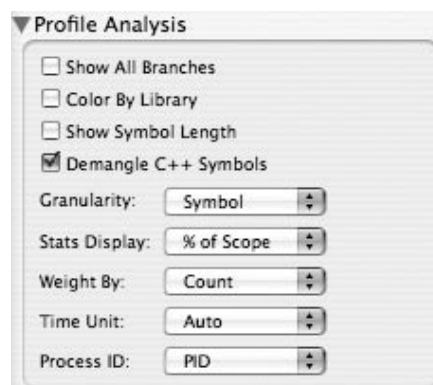
If you choose to display a value instead of a percentage, the Weight By pop-up menu comes into play. From the Weight By pop-up menu, you can tell Shark to display the value as the number of samples or as the amount of time. Only the Self and Total columns can display the amount of time. If you choose Time from the pop-up menu, Shark displays the number of each performance event you told Shark to record.

The Granularity pop-up menu lets you decide how Shark groups samples. The granularity level determines what a row in the Shark results window represents. Initially the granularity level is Symbol, which means there's one row for each function. Other levels of granularity include the following:

- Address granularity, where Shark groups samples by instruction address. A function can appear in multiple rows when you choose address granularity.
- Library granularity, where Shark groups samples by code library. Library granularity generates one row in the results window for each library in your program. The Symbol column disappears when you choose library granularity.
- Source file granularity, where Shark groups samples by source code file. Source file granularity generates one row in the results window for each source code file in your program. Code you didn't write has the file name `NO_SRC_FILE`.
- Source line granularity, where Shark groups samples by source code file and line number. Source line granularity generates one row in the results window for each line of code in your program. Code you didn't write has the file name `NO_SRC_FILE`. Source line granularity lets you pinpoint lines of code that are slowing down your program.

## Viewing Charts

Clicking the Chart tab in the results window changes the results window to show charts. The charts let you examine individual samples. If you wanted to, you could examine every sample chronologically.



**Figure 6.13**

Profile analysis drawer.

## Call Stack Chart

Every Shark profile has at least one chart, which plots the call stack depth for each sample. Sharp spikes in the graph indicate your program was busy when Shark took that sample. A squiggly line signifies a context switch, which occurs when the operating system switches threads.

Initially the call stack chart type is Saturn, which means Shark draws a continuous bar graph, with one bar for each sample. In the Saturn chart type, user call stacks are blue and kernel call stacks are red. If you open the advanced settings drawer, shown in Figure 6.14, you can change the appearance of the call stack chart. Use the Type pop-up menu. There are three additional options.

- Trace, where Shark draws a line graph and does not differentiate between user and kernel call stacks.
- Delta, where Shark plots the change in call stack depth from the previous sample instead of plotting the call stack depth.
- Hybrid, which is a combination of Saturn and Trace. It plots a line like Trace, but when you select a sample, it fills a block like Saturn.

## Performance Event Charts

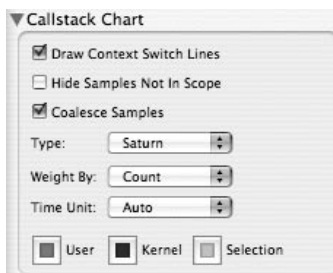
In addition to the call stack depth chart, there's one chart for each performance event you told Shark to record. Shark plots the number of each event that occurred for each sample. If you don't see performance event charts, choose File > Show Advanced Settings. In the Performance Count Data Mining section, you'll see the performance events Shark counted. Select the left checkbox (the one under the eye column) for an event to tell Shark to show the chart for that event.

If you open the advanced settings drawer, there's a sum column with a checkbox for each performance event Shark recorded. Selecting the sum checkbox plots the event as a running total with each sample adding to the previous samples. Plotting as a running total gives the graph a gradual upward slope as you move from left to right.

## Viewing Individual Samples

Initially Shark fits the whole graph in the window. Fitting all the samples in the window provides an overview, but makes looking at an individual sample difficult. The slider below the charts zooms the graphs in and out, allowing you to focus on certain samples.

Clicking a chart fills the right side of the window with the call stack Shark recorded for that sample. Double-clicking a function in the call stack displays that function's source code in the results window. Use the arrow keys to navigate the samples. Pressing the right arrow key moves you to the next sample, and pressing the left arrow key moves you to the previous sample.



**Figure 6.14**

Call stack chart advanced settings.

For most charts clicking inside a chart draws a yellow line where you clicked. The yellow line denotes the current sample. The exceptions are the Saturn and Hybrid types of the call stack depth chart. In these cases clicking the chart colors the sample's call stack pyramid yellow. The top of the pyramid is the function at the top of the call stack and the base of the pyramid is the first function your program called.

## Data Mining

Shark provides you with a staggering amount of information, making it difficult to find the information you need. Shark provides data mining capabilities to help you find what you need.

Choose File > Show Advanced Settings to open the data mining drawer, which you can see in Figure 6.15. The Apply to Heavy and Apply to Tree checkboxes let you specify the views where the data mining takes effect. Selecting only one of the checkboxes lets you have data mining in one view and no data mining in the other view.

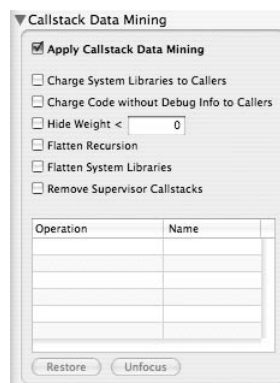
### Excluding Portions of Code

The easiest way to find the information you want is to exclude the information you don't want. Excluding a function hides it in the Shark window and passes its costs to the functions that called it. Suppose function A calls function B and your program spent one second in A and three seconds in B. If you exclude B, Shark says your program spent four seconds in A, the one second the program spent in A plus the three seconds it spent in B.

### Excluding Areas of Code

There are a series of checkboxes in the Data Mining section of the advanced settings drawer that let you exclude large areas of your code. Selecting the Charge System Libraries to Callers checkbox tells Shark to pass the calls your code makes to system libraries to the functions in your code that made the calls. Passing the calls to your functions lets you see the functions in your code where your program is spending the most time. With this checkbox unselected, Shark reports lots of lower level function calls you didn't write. Selecting the checkbox moves the time spent in these lower level calls to functions you wrote.

Selecting the Charge Code without Debug Info to Callers checkbox hides the functions that don't have source code available. In most cases, selecting the Charge Code without Debug Info to Callers checkbox excludes the code you didn't write, allowing you to focus on your code.



**Figure 6.15**

Data mining drawer.



If eliminating all the code you didn't write goes too far, the Remove Supervisor Callstacks checkbox may be for you. Selecting the Remove Supervisor Callstacks checkbox excludes functions in supervisor processes from Shark's display.

The Hide Weight < text field lets you exclude less important functions. How Shark excludes the functions depends on what you choose in the Weight By menu. If you weight by count, Shark excludes functions whose sample count is less than what you enter in the text field. If you weight by time, Shark excludes functions whose time is less than what you enter in the text field. The text field uses seconds as the unit of measurement. If you want to exclude functions where your program spent less than 25 milliseconds, enter the value .025 in the text field.

## Excluding Individual Functions and Libraries

There are two ways to exclude a function. Selecting a function and choosing Data Mining > Charge to Callers removes the function from the results window and passes the costs to the functions that call this function. Selecting a function and choosing Data Mining > Remove Callstacks with Symbol removes the call stacks where the function appears. Removing call stacks changes the percentages Shark displays in the results window because the total number of samples is lower due to the removal of the call stacks.

Choose Data Mining > Charge Library to Callers to charge calls made to a library's functions to the callers. If your program spends a lot of time in functions of a library you didn't write, charging the calls to that library lets you see the areas of your code that are responsible for consuming so much time.

When you exclude a library or function, Shark adds it to the list of excluded names. To restore the library or function, select it from the list and press the Delete key. The Restore All button restores all the excluded libraries and functions.

## Flattening Libraries

Flattening a library excludes all the functions in a library except for its entry points. The costs of the excluded functions go to the entry points. A library's entry point is a function that is available to other programs. Apple's libraries use entry points for your program to enter the libraries. The entry points are functions your program calls to get into a particular library.

To flatten one library, select one of its functions from the Shark window and choose Data Mining > Flatten Library. The Flatten System Libraries checkbox lets you flatten all the system libraries your program links to. The Flatten Recursion checkbox tells Shark to flatten recursive function calls. When a function calls itself, the function call is recursive. Selecting the Flatten Recursion checkbox places all calls to the recursive function in one call tree listing, allowing you to focus on the recursive function itself, not each individual call to the function.

## Focusing on Functions

Excluding functions and flattening libraries hide the information you don't need, but focusing lets you concentrate on the functions you're interested in. There are two ways to focus: focus on a function or focus on a function's callers. Focusing on a function sets the root of the call tree to the focused function, allowing you to concentrate on the routines the focused function calls. Focus on a function by selecting it and choosing Data Mining > Focus Symbol.

Focusing on a function's callers sets the leaf of the call tree to the function, allowing you to see which routines call the function. Select a function and choose Data Mining > Focus Callers of Symbol.

## Performance Event Data Mining

If you recorded performance events in your Shark session, you can data mine the events. Choose File > Show Advanced Settings to open the advanced settings drawer. Go to the Performance Count Data Mining section, which you can see in Figure 6.16. Select the Apply Perf Count Data Mining checkbox to turn on data mining for performance events.

Each performance event Shark was recording has a menu in the Remove column. This menu lets you specify a condition. Shark excludes samples that meet the condition. The Value column is where you specify the value of a condition. If you wanted to exclude samples where a performance event happened fewer than 1000 times, you would choose < from the Remove menu and supply the number 1000 in the Value column.

## System Trace Results

The system trace profile's output differs from the other Shark profiles so I've placed the material here. The results window for the system trace profile has three tabs.

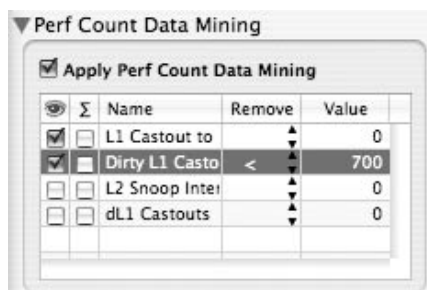
- Summary.
- Trace, which lets you examine every sample Shark took during the system trace.
- Timeline, which provides a chart of the samples Shark took during the system trace.

Use the Process pop-up menu to choose a program to view. Use the Thread pop-up menu to choose a thread to view. If you have a dual processor Mac, use the CPU pop-up menu to choose a CPU to view.

## Summary

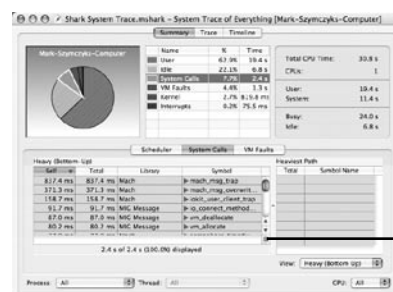
The summary view, shown in Figure 6.17, is the initial view Shark shows. It provides general statistics about the trace. At the top of the window Shark divides the trace into the following categories:

- The time spent in the applications you're viewing in the trace.
- The time spent in operating system calls.
- The time spent in virtual memory faults.
- The time spent in interrupts.
- The time spent in the operating system kernel.
- The time the CPU was idle.



**Figure 6.16**

Performance event data mining.



**Figure 6.17**

System trace summary.

Call Stack Button

Shark shows the breakdown as a table and as a pie chart. Next to the breakdown Shark shows the total time spent in the trace. It breaks the total time into the time spent in applications and the time spent in operating system functions. Shark also divides the total time into the time the CPU was busy and the time the CPU was idle.

In the center of the window are three tabs: Scheduler, System Calls, and VM Faults. Use these tabs to view statistics about the scheduling intervals, system calls, and virtual memory faults in the system trace.

## Scheduler Summary

When you click the Scheduler tab in the summary view, Shark tells you the following statistics for each program:

- The number of scheduling intervals.
- The total amount of time.
- The average amount of time per interval.
- The minimum amount of time in one interval.
- The maximum amount of time in one interval.

Underneath the statistics is a pop-up menu. The initial value is Busy Time, which means the scheduler reports statistics for the time the CPU wasn't idle. Busy time can be separated into user time and system time. User time is the amount of time spent in applications, and system time is the amount of time spent in operating system calls. The pop-up menu has items to view scheduling statistics for user time and system time. Choosing Priority from the pop-up menu shows you the average, minimum, and maximum scheduling priorities for each program.

Clicking the disclosure triangle next to a program shows you the statistics for the program's threads.

## System Calls and VM Faults Summaries

Clicking the System Calls tab shows statistics about the operating system calls made during the trace. Clicking the VM faults tab shows statistics about the virtual memory faults that occurred during the trace. When you click the System Calls or VM Faults tabs, the results window looks more like the results window for other profiles. There are Self, Total, Library, and Symbol columns. There are heavy and tree views. There's a call stack button to show the call stack in the results window.

Shark initially groups the system calls and VM faults by name. If your program makes a system call 1000 times, Shark collects the 1000 calls into one listing. Choose View > Show Advanced Settings to open the advanced settings drawer. Deselect the Group by checkbox to turn off grouping. When you turn off grouping, the call tree looks more like a regular Shark call tree.

## Trace

Click the Trace tab in the results window to display the trace view. The trace view provides more detailed statistics about the scheduler, system calls, and VM faults. You can examine every sample Shark took during the system trace.

**Scheduler Trace**

Clicking the Scheduler tab in the trace view shows the following information for each scheduling interval:

- Index, which is the sample number.
- Process, which is the name of the program.
- Thread.
- Time, which is the total amount of time in the scheduling interval.
- User Time, the time spent in applications.
- System Time, the time spent in operating system calls. The sum of the User Time and System Time columns equals the Time column.
- Delta Time, which is the amount of time that elapsed between this sample and the previous sample Shark took for this thread.
- Reason, which is the reason the operating system interrupted the program. The most common reasons are preemption, giving time to another program, and blocked, waiting for an event to occur.
- Priority, which is the scheduling priority. A higher number indicates higher priority.

**System Calls Trace**

Clicking the System Calls tab in the trace view shows the following information for each system call:

- Index, which is the sample number.
- Interval, which tells you the scheduling intervals when the system call was active.
- Process, which is the name of the program.
- Thread.
- Name, which is the name of the system call.
- Return Value, which is what the system call returns.
- CPU Time, which is the time the CPU spent in the call.
- Wait Time, which is the time the process waited for the system call to complete.

Below the system call are five text fields that contain arguments. Selecting a system call from the list fills the text fields with the selected system call's arguments.

**VM Faults Trace**

Clicking the VM Faults tab in the trace view shows the following information for each virtual memory fault:

- Index, which is the sample number.
- Interval, which tells you the scheduling intervals when the fault occurred.
- Process, which is the name of the program.
- Thread.
- Type, which is the type of fault. The most common types are cache hits and page ins.
- CPU, which is the amount of time the CPU spent in the fault.
- Wait, which is the amount of time the process waited for the fault to complete.
- Library, which is the library where the fault occurred.
- Address, which is the memory address where the fault occurred.
- Size. The size is usually 4 KB because PowerPC memory pages are 4 KB.

## Timeline

The timeline, shown in Figure 6.18, provides a chart of the system trace. The left side of the window contains a list of threads. The right side of the window contains the chart. Bars of color indicate the CPU scheduled an interval for the thread. No color means the CPU was in another thread.

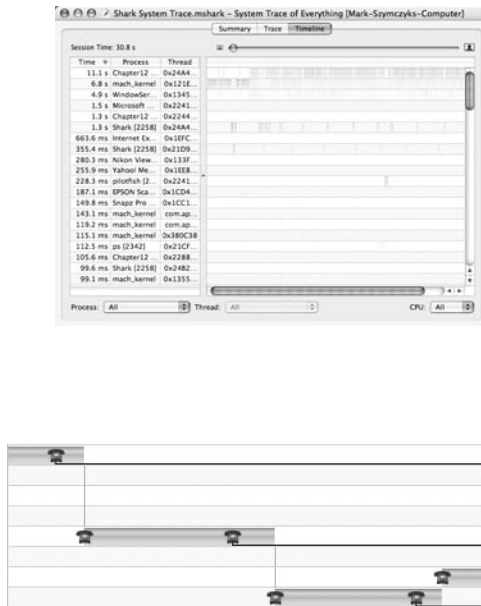
If you want to walk through the entire trace, tell Shark to look at all processes and drag the slider at the top of the window to the right edge. Dragging the slider zooms the chart so you can see individual scheduling intervals. When you have the chart zoomed in, Shark draws lines to show the CPU giving time to another thread. Refer to Figure 6.19.

When you have the chart zoomed in, you'll see two types of icons. A telephone icon indicates a system call. A document icon indicates a VM Fault. Clicking an icon in the chart displays the call stack for the system call or VM fault.

## MONster Profile

If your profile uses the MONsterProfileAnalysis and MONsterProfileViewer plug-ins, the Shark results window has a Counters tab. Figure 6.20 shows the counters view. At the top of the window is a table of data. There is one column for each performance event you're counting and each profiling equation. There is one row for each sample Shark took plus rows for the total, average, geometric mean, minimum, and maximum.

Selecting a column heading displays a graph of that column's data. Use the slider at the bottom of the window to zoom in on specific samples.

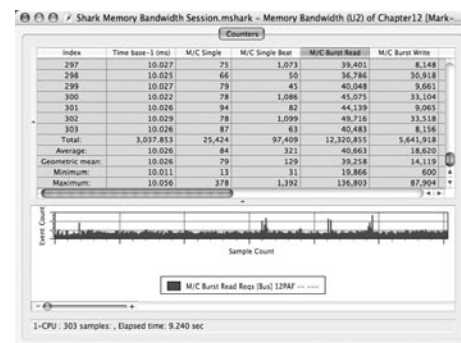


**Figure 6.19**

The timeline showing the operating system giving time to threads.

**Figure 6.18**

System trace timeline.



**Figure 6.20**

The counters view for profiles that use the MONsterProfileViewer plug-in.

## Saturn

Saturn is a function-level profiler. You can think of Saturn as a graphical version of `gprof`, which I covered in Chapter 5. Like `gprof`, Saturn tells you the amount of time your program spends in each function. Plus, Saturn can record a Performance Monitor Counter (PMC) event as your program runs. The PMC events it can record are CPU cycles, instructions completed, and cache misses on data in the Level 2 (L2) cache.

### Before You Profile

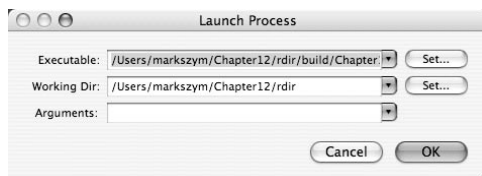
The Saturn preferences panel lets you specify the PMC event you want Saturn to record. Choose Saturn > Preferences to display the preferences panel. After choosing the PMC event Saturn records, you must tell Saturn whether to count the PMC events for just your program or for all programs. Choose Per Process to tell Saturn to count only the PMC events generated by your program.

Should you choose to record a PMC event, remember that recording PMC events increases the size of the data file Saturn creates, which means Saturn takes more time to load the data file. Macs have hundreds of millions of CPU cycles per second, and Mac programs execute hundreds of thousands of instructions. Saturn keeps track of the call stack for each CPU cycle, instruction completed, or L2 data cache miss, resulting in lots of call stack listings. If you want Saturn to load the profile data quickly, don't record PMC events.

Saturn takes `gprof`'s `gmon.out` files as input. To profile your program in Saturn, you must compile it to generate profiling code. Refer to the section "Generating Profiling Code in Xcode" in Chapter 5 for detailed instructions on generating profiling code.

### Profiling Your Program

Choose Saturn > Launch Process to run your program in Saturn. A dialog box like Figure 6.21 opens. Click the Set button next to the Executable combo box to select the program you want to run. The working directory is where Saturn places its results. If you don't set a working directory, you may have a hard time finding the results. Click the Set button next to the Working Directory combo box to tell Saturn where you want to place the results. Click the OK button to launch your program. Run your program the way a user typically would and quit the program. A folder with a name starting with `Saturn_profile` should appear where you set the working directory.



**Figure 6.21**

Saturn dialog box to launch a program.

Function Name	Count	Self Time	Total Time	Self PMC	Total PMC
main	1	0.00 (0.00%)	42.25 (100.00%)	39	38365719
GameApp::EventLoop()	1	12.32 (29.18%)	38.47 (91.04%)	5573027	33797912
GameApp::GameLoop()	399	0.02 (0.04%)	24.78 (58.65%)	41298	26974290
GameApp::HandleEvent(EventRecord*)	12	0.00 (0.00%)	1.37 (3.24%)	72	1250595
GameApp::InitApp()	1	0.00 (0.00%)	3.77 (8.93%)	1189	4531096
GameApp::CleanupApp()	1	0.01 (0.02%)	0.02 (0.04%)	15532	36396
GameApp::GameApp(in::charge)	1	0.00 (0.00%)	0.00 (0.00%)	0	276

**Figure 6.22**

Saturn results window.

## Viewing Saturn's Results

When you quit your program after running it in Saturn, an Open File dialog box appears asking for a Saturn data file to open. Go to the folder where you told Saturn to store the results and open the file with the `.sat` extension. The data file Saturn creates can be huge, which means Saturn can take several minutes loading the data for you to view.

The top half of the Saturn window, which you can see in Figure 6.22, lists the call tree of functions your program called. For each function Saturn tells you the following information:

- The function name.
- Count, the number of times your program called the function.
- Self Time, the amount of time spent in the function.
- Total Time, the amount of time spent in the function and its descendants, the routines below the function in the call tree.
- Self PMC, the number of PMC events recorded for this function. The PMC event type is the one you set in the preferences panel.
- Total PMC, the number of PMC events recorded for this function and its descendants.

Initially Saturn shows only the first function your program called. Click the disclosure triangle to reveal more of the call tree.

Double-clicking a function opens a call history window for the function. The call history window has one listing for each time your program called the function. In each listing the window tells you the amount of time your program spent during this particular call and the number of PMC events that occurred.

The bottom half of the Saturn window shows a graph listing the call stack depth when each PMC event occurred. Initially the graph shows the entire graph. Use the slider at the bottom of the window to zoom in on a group of samples.

## CHUD Command-Line Tools

Shark and Saturn are great for finding the slow functions in your code. What they don't tell you is why the functions are slow. To learn why a function is slow, use the suite of command-line tools that are part of the CHUD Tools package. The first command-line tool you will use is `amber`. `amber` creates a trace file that records your program's actions. Pass the trace file to the other command-line tools: `simg4`, `simg5`, and `acid`.

The command-line tools work on two levels. First, they generate lots of statistics about your program. Second, they list every machine instruction your program executed during the trace. Because the command-line tools generate so much data, you should trace one function instead of your whole program. Tracing an entire program will give you a staggering amount of data.

The information the command-line tools generate is low-level, which can be difficult to understand. The more you know you about PowerPC assembly language and PowerPC processors, the more you'll get out of the command-line CHUD tools.

**amber**

**amber** records every machine instruction your program executes and every data access your program makes as the program runs. It saves the recorded data to a trace file, which you pass to `simg4`, `simg5`, and `acid`.

The easiest way to run **amber** is to use the `-a` option and supply your program's process ID. Run the `top` program from Terminal to find the process ID.

```
amber -a ProcessID
```

The previous listing tells **amber** to trace the entire program. To trace a single function, use the `-B` and `-E` options. The `-B` option tells **amber** to start tracing when your program enters a function that you specify, which would be the function you want to trace. The `-E` option tells **amber** to stop tracing when your program enters a function that you specify, which would be the routine that calls the function you want to trace.

```
amber -a ProcessID -B FunctionToTrace -E FunctionToStop
```

C++ functions are more complicated to trace with **amber**. Multiple C++ functions can share the same name. The compiler mangles the function names to give each function a unique name. You must supply the mangled function names to **amber**. To find the mangled function names, open the object file where the function resides. You can find the object files in the following directory in your project folder:

```
build/ProjectName.build/Debug/ProjectName.build/Objects-normal/ppc
```

On Intel Macs the last directory will be `i386` instead of `ppc`. Release builds will have a `Release` directory instead of a `Debug` directory.

There is one object file for each source code file in your project. The object file has the extension `.o` or `.ob`. Open the object file in a text editor. Search for a function name to find its mangled name. The `gcc` compiler normally mangles C++ function names by giving each function name the prefix `ZN` and the suffix `Ev`. It also inserts a number between the class name and the function name. Include everything from `ZN` to `Ev` when passing a mangled function name to **amber**.

Running **amber** doesn't do anything until you tell it to start tracing. Press Control-Esc to start a trace, and press Control-Esc to stop the trace. If you quit **amber** without stopping the trace, **amber** doesn't save any information. When you stop a trace, **amber** reports the number of instructions traced and the amount of time spent tracing. Run as many traces as you want. Press Option-Esc to quit **amber**.

When you quit **amber**, the current directory has one folder for each trace you ran. The name of the folder is `trace_XXX`, where `XXX` is a three-digit number. Inside a trace folder is one trace file for each thread in your program. The file name is `thread_XXX.tt6e`, where `XXX` is a three-digit number. The trace files are what you pass to `simg4`, `simg5`, and `acid`.



## simg4

simg4 takes an `amber` trace file and simulates the way the instructions in the trace file would run on a PowerPC 7410 processor, which is especially handy if your Mac doesn't have a G4 processor.

Running `simg4` takes the following form:

```
simg4 [Options] < [Input File] > [Output File]
```

simg4 has many options and many runtime parameters you can use to customize the simulation, too many to cover right now. I will cover some of the more important parameters and options later. Refer to *Sim\_G4 User's Guide*, which is part of the CHUD Tools documentation, for a complete list of options and runtime parameters.

*Input File* is a trace file `amber` created for you. *Output File* is where `simg4` writes the results. If you don't supply an output file, `simg4` writes its results in the shell window.

```
simg4 < thread_001.tt6e > simg4Results.txt
```

## Output

The output file `simg4` creates contains many statistics, so many that I've decided to break them down into their individual sections.

### Instruction Flow Statistics

The first step the CPU takes with an instruction is fetching the instruction. The CPU fetches the instruction from cache or from RAM if the instruction isn't in any of the G4's caches. Most fetched instructions move to the dispatch stage. In the dispatch stage the CPU dispatches the instruction to the appropriate execution unit. Once dispatched the instruction executes. After executing, the instruction moves to the completion queue. When the instruction leaves the completion queue, the instruction is retired.

Branch instructions are more complicated. They contain two paths of instructions: one if the CPU takes the branch and one if the CPU does not take the branch. If the CPU does not take the branch, it removes the branch instruction from the instruction stream and continues fetching instructions.

If the CPU takes the branch, it takes the instructions that make up the branch not taken path and folds those instructions out of the instruction stream. The instructions for the branch's target, where the branch moves the program, replace the folded instructions. When the CPU takes the branch, the branch instruction moves to the dispatch stage like non-branching instructions do.

simg4 reports the following statistics regarding instruction flow:

- The number of instructions fetched.
- The number of instructions dispatched.
- The number of instructions retired.
- The number of instructions folded.

## Dispatch Stalls

A *stall* occurs when an instruction cannot move to the next stage of execution. The dispatch stalls section provides information on the stalls that occur when dispatching instructions to the execution units. For dispatch stalls `simg4` lists the percentage of the total clock cycles stalled with the number of stalled cycles in parentheses. `simg4` alerts you to the following dispatch stalls:

- Drain, which occurs when the CPU incorrectly predicts a branch. On an incorrectly predicted branch, the CPU drains the branch and any accompanying instructions from the instruction queue.
- IB empty, which means there are no instructions to dispatch.
- CB full, which means the completion buffer is full. The completion buffer is where the CPU places instructions when it is finished with them.
- GPR\_rename, which means the CPU has run out of integer rename registers.
- FPR\_rename, which means the CPU has run out of floating-point rename registers.
- VR\_rename, which means the CPU has run out of vector rename registers.
- Unit busy, broken down by execution unit. An execution unit is busy when it's executing an instruction.
- Dispatch serialization instructions, which allow the CPU to dispatch and complete only one instruction in a clock cycle instead of the two instructions the G4 can normally dispatch.
- Tail serialization instructions, which force the CPU to wait until the tail serialization instruction finishes before dispatching a new instruction.

Let's use an example to further explain drains. Suppose you have a `while` loop in your program. The CPU predicts the program will enter the loop because the condition in the `while` loop can be false only once, but can be true multiple times. To make each trip through the `while` loop as fast as possible, the CPU places the `while` loop's instructions in the instruction queue along with the branch. When the condition in the `while` loop becomes false, the CPU must flush the `while` loop's instructions from the instruction queue.

Rename registers deserve more explanation. The CPU uses rename registers to store the results of an instruction when dispatching the instruction to an execution unit. Rename registers let the CPU execute instructions out of order. Executing instructions out of order lets the CPU execute multiple instructions simultaneously, which makes your program run more efficiently. When an execution unit executes an instruction before it's needed, the results of the instruction go into a rename register. The G4's integer, floating-point, and vector units each have six rename registers. The PowerPC 7450 has 16 rename registers. When an execution unit runs out of rename registers, the CPU must wait, causing a stall.

## Execution Unit Statistics

`simg4` reports statistics on the following execution units:

- Two fixed point (integer) units.
- Floating point unit.
- Vector simple integer unit. The simple integer unit can handle only simple integer instructions that execute in one cycle.
- Vector complex integer unit. The complex integer unit handles integer instructions that take longer than one cycle to execute, such as division.
- Vector float unit.
- Vector permute unit, which handles bit shifts and permute instructions.
- System register unit.
- Load/store unit.

For each execution unit `simg4` reports the following information:

- The percentage of time the unit was idle.
- The number of instructions dispatched to the unit.
- The number of stalls caused by data dependencies. A *data dependency* occurs when an instruction, instruction A, relies on the results of another instruction, instruction B. If instruction B has not finished executing, instruction A stalls, waiting for B to finish.
- The number of stalls caused by serializing instructions.

## Retirement Stalls

The retirement stalls section provides the following information on stalls that occur when retiring an instruction:

- The percentage of instructions that stalled because of too many condition register updates. The PowerPC 7410 processor can retire two instructions in one clock cycle. If the two instructions to retire update the condition register more than two times, a stall occurs.
- The percentage of instructions that stalled because of too many rename register updates. These percentages are broken down into integer, floating-point, and vector rename registers. If the two instructions to retire update an execution unit's rename registers more than two times, a stall occurs.
- The percentage of instructions that stalled because the CPU retired the maximum number of instructions in a clock cycle. The PowerPC 7410 processor can retire two instructions in one cycle. If more instructions are eligible to retire, they stall.
- The percentage of instructions that stalled due to unresolved branches having to wait for older branches to resolve. A branch is unresolved when the CPU does not know if it should take the branch. `sim_g4` reports the percentage as `retire_spec`.
- The percentage of instructions that stalled waiting for older instructions to retire. On G4 processors instructions can execute out of order, but must retire in order. `simg4` reports the percentage as `retire_bottom`.

## Branch Statistics

`simg4` reports three levels of branch statistics. First, `simg4` reports the number of branches encountered and tells you the following information:

- Percentage of branches taken.
- Percentage fall through, which is the percentage of branches not taken.
- Percentage folded.
- Branch prediction rate.
- Percentage of branches always taken.

Next comes a series of branch stalls, listed in Table 6.2. For each type of stall, `simg4` reports the percentage of total clock cycles the CPU had to wait because of the stall.

Finally comes the total number of branch target instruction cache (BTIC) hits and the hit rate. The BTIC contains the most recent branch target instructions. When the CPU takes a branch, loading the branch target's instructions goes faster if the instructions are in the BTIC.

Table 6.2 Branch Stalls

Stall	Description
ctr busy stall	If a branch that uses the count register follows an instruction that moves data to the count register, the branch must wait for the move instruction to complete.
lr busy stall	If a branch that uses the link register follows an instruction that moves data to the link register, the branch must wait for the move instruction to complete.
ctr max stall	The PowerPC 7410 allows only two unresolved branches based on the count register at one time. A third branch based on the count register must wait until the CPU resolves one of the two branches.
lr max stall	The CPU currently has the maximum number of unresolved branches based on the link register. Additional branches based on the link register must wait until the CPU resolves one of the branches.
out of cr ports stall	On G4 processors the condition register consists of eight 4-bit fields. When other instructions are using all eight fields, a stall occurs.
max spec branches	There were too many speculative branches. The PowerPC 7410 can have two speculative branches at one time. <i>Speculative branches</i> are unresolved branches.
max spec_stream exec	The CPU executed the maximum number of speculative streams. The PowerPC 7410 can execute two speculative streams. An unresolved branch has two paths of instructions. Each of these paths is a speculative stream.
Max branches per cycle	The CPU executed the maximum number of branches it could execute in one clock cycle.
Br_taken bubble stall	When the CPU takes a branch, it must load the instructions for the branch's target, where the branch moves the program. There is a one or two cycle delay to load the instructions. This delay is a bubble.
End_of_cache_line stall	The branch instruction appears at the end of one cache line, and the target instruction appears at the start of a second cache line. This causes the CPU to load an extra cache line.
Mispredicting stall	The CPU incorrectly predicted the branch so the CPU must flush the instructions in the incorrectly predicted path from the instruction stream.
Flush serialization	When the CPU takes a branch, it must flush the instructions in the branch not taken path out of the instruction stream. Flushing the instructions causes a stall.

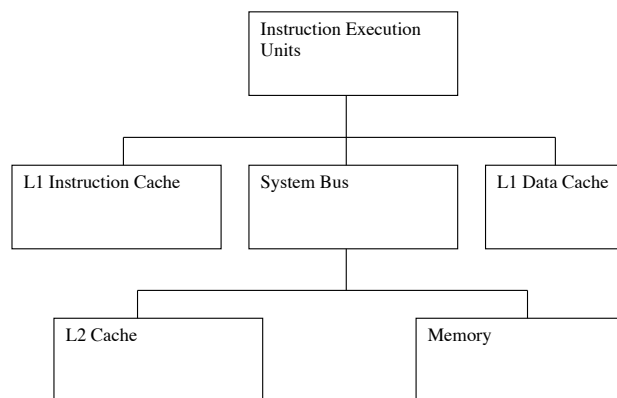


Figure 6.23

PowerPC cache diagram.

## L1 Instruction Cache Statistics

I should start by explaining what the L1 instruction cache does. *Cache* is high speed memory. On PowerPC processors, the fastest cache is the Level 1 (L1) cache, which consists of the instruction cache and the data cache. The L1 instruction cache stores recently used instructions. When your program must execute an instruction that's in the cache, the CPU fetches it from the cache and dispatches it to the appropriate execution unit.

Looking at Figure 6.23, you can see why loading the instruction from the L1 instruction cache is much faster than loading it from the Level 2 (L2) cache or memory. The instructions in the L1 instruction cache can move directly to an execution unit without having to travel along the processor (system) bus. The processor bus runs much slower than the CPU: 100 to 167 MHz for G4 Macs, one third the CPU clock speed for the iMac G5, and one third or one half the CPU clock speed for G5 Mac towers.

First, `simg4` reports characteristics of the L1 instruction cache.

- Cache size.
- Cache line size. A *cache line* is the smallest block of memory the CPU can copy to the cache.
- Associativity, which describes the number of sets the cache memory space is divided into. An 8-way associative cache has its memory space divided into eight sections. Lower associativity is better.
- Whether or not the cache was preloaded when running `simg4`. If the cache was preloaded, `simg4` treats accesses to invalid entries as if they were valid.
- Reload penalty, which is the number of clock cycles required to reload a cache line when the CPU cannot find an instruction in the L1 and L2 caches.

To preload the caches and translation lookaside buffers, run `simg4` with the `-r` option and set the runtime parameter `warmup` to the value 1.

```
simg4 -r warmup=1 < thread_001.tt6e > simg4Results.txt
```

After the cache characteristics come the number of cache hits and cache misses. A *cache hit* occurs when the CPU finds what it needs in the cache, which is an instruction in the case of the L1 instruction cache. A *cache miss* occurs when the CPU can't find what it needs in the cache. When the CPU can't find an instruction in the L1 instruction cache, it looks in the L2 cache. If the instruction isn't in the L2 cache, the CPU loads the instruction from RAM.

Finally, `simg4` reports the number of cache invalidations. A *cache invalidation* occurs when a cache line no longer contains a valid value. The invalid value must be removed from the cache. The instruction cache's cache lines contain memory addresses of instructions. If the contents of one of these memory addresses changes, the cache has an invalid value, causing a cache invalidation.

## ITLB Statistics

The Instruction Translation Lookaside Buffer (ITLB) saves recent page address translations for instructions. A *page address translation* calculates the physical address for a page of memory from its effective address. Each running Mac OS X program has 4 GB of effective address space on a G4 processor. There's not enough physical memory to store 4 GB for each running program. Because there's less physical memory than effective memory, the CPU must perform page address translations.

Storing recent page address translations in a translation lookaside buffer keeps the CPU from having to perform unnecessary page address translations. Avoiding unnecessary page address translations makes instruction and data accesses faster.

`simg4` reports the following information for the ITLB:

- The number of entries the buffer holds.
- Associativity.
- Whether or not the ITLB was preloaded when running `simg4`. If the ITLB was preloaded, `simg4` treats accesses to invalid entries as if they were valid.
- Buffer hits.
- Buffer misses. An ITLB miss occurs when there are no instructions to dispatch or retire.

### **L1 Data Cache Statistics**

The L1 data cache stores the most recently used data. Loading data from the L1 data cache allows the data to move directly to the appropriate execution unit. `simg4` reports the following characteristics of the L1 data cache:

- Cache size.
- Cache line size.
- Associativity.
- Whether or not the cache was preloaded when running `simg4`. If the cache was preloaded, `simg4` treats accesses to invalid entries as if they were valid.

`simg4` reports the following statistics for the L1 data cache:

- Cache hits, broken down into loads and stores.
- Cache misses, broken down into loads and stores.
- Cache hit rate.
- Reloads, which occur in the L1 data cache when the CPU can't find the data in the L1 and L2 caches.
- Castouts, which are cache lines that have been replaced by new blocks of data. Data cast out of the L1 data cache moves to the L2 cache.
- Percentage of the cache's bandwidth (BW) used.

### **DTLB Statistics**

The Data Translation Lookaside Buffer (DTLB) saves recent page address translations for data accesses. First, `simg4` reports the characteristics of the DTLB.

- The number of entries the buffer holds.
- Associativity.
- Whether or not the DTLB was preloaded when running `simg4`. If the DTLB was preloaded, `simg4` treats accesses to invalid entries as if they were valid.

The DTLB's statistics follow the characteristics.

- Buffer hits, broken down into loads and stores.
- Buffer misses, broken down into loads and stores. A DTLB miss occurs when a load or store instruction is the next instruction to be retired.
- Stalls.
- Hit rate.

## Software Prefetching Statistics

*Prefetching* is retrieving data and instructions before they execute so the CPU pipeline stays full. Software prefetch occurs when the software initiates a prefetch instead of the hardware.

On PowerPC processors the data stream touch (`dst`) series of instructions is responsible for software prefetch. The `dst` instructions initiate a cache prefetch from software. The prefetched data and instructions come from the data streams in the Altivec unit. Up to four data streams can exist simultaneously. `simg4` breaks the `dst` instructions down by data stream.

## L2 Direct Mapped SRAM Statistics

G4 chips let you specify part of the L2 cache SRAM as direct-mapped memory. A *direct-mapped cache* allows a memory address to appear in only one location in the cache. Cache hits in direct-mapped caches execute faster.

To see any statistics for direct-mapped memory, you must enable direct-mapped memory when you run `simg4`. To enable direct-mapped memory, run `simg4` with the following option:

```
-r l2_dm_sram_enable=1
```

Turning on direct-mapped memory makes the entire L2 cache direct-mapped. To specify a smaller size, use the `l2_dm_sram_size` runtime parameter. The following command runs `simg4` with 128 KB of the L2 cache direct-mapped:

```
simg4 -r l2_dm_sram_enable=1 l2_dm_sram_size=128 < thread_001.tt6e >  
simg4Results.txt
```

`simg4` reports the following information for L2 direct-mapped SRAM:

- The starting address of the direct-mapped memory in the L2 cache.
- The ending address.
- The amount of direct-mapped memory.
- The number of accesses to the direct-mapped memory, broken down into instruction, data, and other accesses.

## L2 Cache Statistics

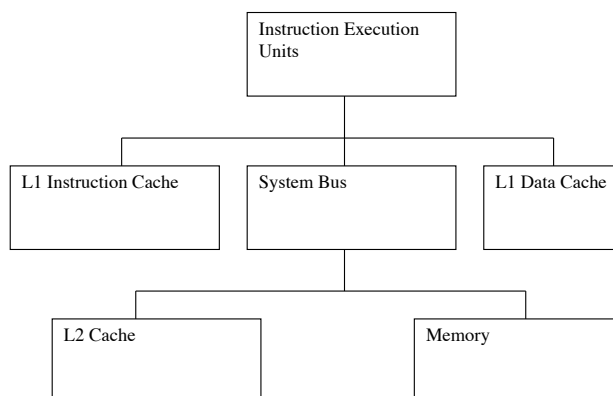
When the L1 instruction and data caches are full and new instructions and data enter the cache, the old instructions and data move down to the Level 2 (L2) cache. L2 cache is high-speed memory, but data and instructions must go through the processor bus to reach the appropriate execution unit, which you can see in Figure 6.24. Going through the processor bus makes the L2 cache slower than the L1 cache, but there's more L2 cache available. The L2 cache holds both instructions and data in one unit.

First, `simg4` reports the L2 cache's characteristics.

- Cache size.
- The number of sectors per tag. Fewer sectors per tag make more efficient use of the cache.
- Sector size, which is the cache block size.
- Associativity.
- Bus ratio, which measures how fast the L2 cache is. A bus ratio of 2:1 means the L2 cache runs at half the clock speed of the CPU.
- SRAM Latency, which is the number of clock cycles it takes to access the first sector during a L2 cache access.
- Data bus size, which can be either 32 or 64 bits.

The L2 cache's statistics follow the characteristics.

- Cache hits, broken down into instructions, data, and writes.
- Cache misses, broken down into instructions, data, and writes.
- Hit rate.
- Reloads. When an instruction misses in both the L1 and L2 caches, the CPU reloads the instruction's cache block to both the L1 and L2 caches. The CPU reloads cache misses on data to the L1 cache only, which means cache misses on instructions are the major cause of L2 cache reloads.
- Castouts, which occur when sectors are replaced in the cache by new instructions and data.
- Percentage of the cache's bandwidth used, separated into the bandwidth used for reads, writes, and turnaround cycles between reading and writing.



**Figure 6.24**

PowerPC cache diagram.



## Processor Bus Statistics

The processor bus statistics give you characteristics and statistics about the processor bus, also known as the system bus, that data and instructions must pass through on their way from L2 cache and RAM to the appropriate execution unit.

The first pieces of information `simg4` reports are the characteristics of the processor bus.

- Memory controller, which can be either MPC 106 or a hypothetical memory controller.
- The number of entries in the data transaction queue (DTQ), which is a list of information about pending data transactions.
- Bus ratio, which is the number of internal clock cycles per bus clock cycle.
- Bus mode, which can be either the 60x bus or the enhanced 60x bus, the MPX bus. The MPX bus provides higher memory bandwidth and works more efficiently on dual processor Macs.
- Data bus size, which is normally 64 bits. The data bus size is the smallest amount of data transferred across the bus.
- DRAM type, which can be either Extended Data Output (EDO) or synchronous DRAM (SDRAM). Most Macs have SDRAM.
- The number of row bits.
- The number of column bits. The number of row and column bits determines the size of a memory device.
- Memory banks. The maximum number of memory banks is eight.
- Number of sub banks, which are internal memory banks used by SDRAM. The number of sub banks may be two or four. On SDRAM the sub banks determine page hits and page misses.
- Latency for page hits, page misses, and page closes. A page close occurs when there's a page miss with SDRAM. The memory bank must be precharged to apply the new sub bank row address. The latency is the number of clock cycles required to access a memory bank.
- Burst latency, which is the latency between beats after the first beat. A beat is a transfer the size of the data bus.
- Memory page size, which is 2 KB for the MPC 106 memory controller. The hypothetical memory controller calculates the memory page size using the number of column bits and a three-bit byte offset that is a part of every memory address. With the default number of 10 column bits, the memory page size would be 8KB, 1KB for the 10 column bits multiplied by 8 for the three bits in the byte offset. The page size determines whether an access to a memory bank is a page hit or a page miss for EDO memory.
- The maximum number of memory pages that can be open at one time.

You can set many of these characteristics as runtime parameters to `simg4` using the `-r` option. Table 6.3 lists the available memory-related runtime parameters. The following example tells `simg4` to use the hypothetical memory controller:

```
simg4 -r mem_controller=2 < thread_001.ttt6e > simg4Results.txt
```

The processor bus statistics follow the characteristics.

- Percentage of the bus bandwidth used.
- Memory page accesses, broken down into reads and writes.
- Memory page hits.
- Memory page misses.
- Memory precharge misses, which occur when accessing a different row in a memory bank.
- Memory pages opened during the course of the simulation.
- Memory pages closed during the course of the simulation.
- Memory open pages, the number of currently opened pages.

**Table 6.3 Processor Bus Runtime Parameters**

Parameter	Default Value	Description
mem_controller	1	The memory controller to use. Use the value 1 for the MPC 106 controller. Use the value 2 for the hypothetical controller.
g4_bus_fraction_numerator	4	The numerator of the bus ratio.
g4_bus_fraction_denominator	1	The denominator of the bus ratio. Use the value 1 for full clock cycles and the value 2 for half clock cycles.
bus_mode	1	Use the value 1 for the MPX bus and the value 0 for the 60x bus. When using the MPC 106 memory controller, the default bus mode becomes the 60x bus.
dram_type	2	The type of memory. Use the value 1 for EDO memory and the value 2 for SDRAM.
dram_row_bits	12	The number of row bits.
dram_column_bits	10	The number of column bits.
edo_miss_latency	8	The latency, measured in clock cycles, for a page miss. Only valid with EDO memory.
edo_hit_latency	4	The latency, measured in clock cycles, for a page hit. Only valid with EDO memory.
edo_burst_latency	3	The latency, measured in clock cycles, for a burst. Only valid with EDO memory.
sdram_bank_bits	2	The number of bits, either 1 or 2, to use to address sub banks. Only valid with SDRAM.
sdram_close_latency	2	The latency, measured in clock cycles, to precharge a memory bank. Only valid with SDRAM.
sdram_miss_latency	11	The latency, measured in clock cycles, for a page miss. Only valid with SDRAM.
sdram_hit_latency	5	The latency, measured in clock cycles, for a page hit. Only valid with SDRAM.
sdram_burst_latency	1	The latency, measured in clock cycles, for a burst. Only valid with SDRAM.
sdram_max_open_pages	2	The maximum number of memory pages that can be open at one time. Only valid with SDRAM.

## Pipe Output

Pipe output gives a play by play account of how your program ran. Use pipe output to examine each machine instruction your program executed. To turn on pipe output, run `simg4` with the `-sp` and `-st` options.

```
simg4 -sp [Output File] -st 0 < [Input File]
```

If you don't want `simg4` to write the output to a file, use a dash as the output file. The `-st 0` tells `simg4` to use a horizontal scroll pipe. Use 1 instead of 0 for a vertical scroll pipe, and use 2 for a wide vertical scroll pipe. A horizontal scroll pipe has one line of output per instruction. A vertical scroll pipe has two lines of output for each clock cycle. A wide vertical scroll pipe has one line of output for each clock cycle.

### Horizontal Scroll Pipe

To look at the pipe output, open the output file in a text editor. The horizontal scroll pipe lets you look at each instruction your program executed. You can see the path each instruction traveled along the instruction pipeline. The example below shows a line of horizontal scroll pipe output. The output is too long to fit on one line of text so I divided it into its six components.

```
61:
90191a20
lwz      R0,0x0(R28)
1211
.....IDER.....
1214
```

The first component of the horizontal scroll pipe output is the instruction number. Instruction numbers start at 0 with a number for each instruction in the trace. The instruction's memory address follows the instruction number. The third component is the instruction itself. The fourth component is the cycle number when the CPU fetched the instruction, and the last component is the cycle number when the CPU retired the instruction, which is when the CPU was finished with the instruction.

The fifth component shows the instruction's path down the pipeline for 30 clock cycles. Table 6.4 lists the possible instruction pipeline states. The example above lists clock cycles 1201-1230. The CPU fetched the instruction in cycle 1211, dispatched it in cycle 1212, executed it in cycle 1213 and retired it in cycle 1214.

### Vertical Scroll Pipe

The vertical scroll pipe lets you examine what the CPU was doing each clock cycle. Each cycle has two lines of output because the G4 processor can execute two instructions in one clock cycle.

```
399  D -   14:( 398) ori   R0,R0,0x4003    MAX
399  - D   15:( 398) cmp   7,R3,R0
```

The first column lists the cycle number, 399, in the listing. The status codes of the two instructions, described in Table 6.4, follow the cycle number. The first code is for the first instruction (the top line) and the second code is for the second instruction (the bottom line). After the status codes comes the instruction numbers, which are 14 and 15 in the listing. In parentheses are the cycle numbers the CPU fetched the two instructions, which are both cycle 398 in the listing. The instructions follow the cycle numbers. The last piece of information in the vertical scroll pipe output is the dispatch status. Table 6.5 lists the possible dispatch status types.

**Table 6.4 Instruction Pipeline States**

State	Code	Description
Not Active	.	The instruction wasn't in the pipeline this cycle. Many instructions execute quickly, under 5 cycles. In a 30 cycle display many instructions won't be active over the 30 cycles.
Overwritten	\	The instruction was in the pipeline longer than 30 cycles. Early cycles were overwritten by later ones to fit into the 30 cycle display.
Fetches	I	The instruction has been fetched from memory and placed in the instruction queue.
Dispatched	D	The CPU dispatched the instruction to the appropriate execution unit.
Executing	E	The instruction is currently executing.
Finished	F	The CPU added the instruction to the completion queue, which means the instruction has finished executing.
Retired	R	The instruction has been removed from the completion queue.
Folded	@	When the CPU encounters a branch instruction, takes the branch, and the branch does not update the count register or link register, the CPU folds the branch instruction out of the instruction stream.

The wide vertical scroll pipe shows the same information the vertical scroll pipe does. The wide vertical scroll pipe combines the two lines of output for each clock cycle into one line.

## simg5

`simg5` takes an `amber` trace file and simulates the way the instructions in the trace file would run on a G5 processor, which is especially handy if your Mac doesn't have a G5 processor.

Running `simg5` takes the following form:

```
simg5 [Input File] [Instruction Count] [CPI Interval]
      [Reset Point] [Output File]
```

*Input File* is a trace file `amber` created for you. *Instruction Count* is the number of instructions you want to run through `simg5`. *CPI Interval* is the number of instructions to wait before `simg5` reports the cycles per instruction (CPI) for this group of instructions. *Reset Point* is the number of instructions to execute before resetting the statistics. Use 1 as the reset point to tell `simg5` not to reset the statistics. *Output File* is the name of the file where `simg5` saves the results. `simg5` gives the file the extension `.results` so you don't have to worry about giving the file an extension. The following command:

```
simg5 thread_001.tt6e 500 25 1 MyTrace
```

Tells `simg5` to use the trace file `thread_001.tt6e`. `simg5` executes 500 instructions. Every 25 instructions it reports the cycles per instruction. The statistics do not reset, and `simg5` saves its results to a file called `MyTrace.results`.

## Output

The output file `simg5` creates contains many statistics, so many that I've decided to break them down into individual sections.

**Table 6.5 Dispatch Status Types**

Status	Code	Description
Insufficient IB Inst	IB	Not enough instructions in the IB unit. This normally occurs when there are no instructions to execute.
Dispatch Serialization	DS	The instruction is a dispatch serialization instruction. Serialized instructions allow the G4 chip to dispatch and complete only one instruction in a clock cycle instead of the normal two.
Completion Buffer Full	DB	The completion buffer is full. The completion buffer is where the CPU places instructions when it is finished with them.
Maximum Rename GPRs Reached	GPR	The CPU has used all its available rename general-purpose (integer) registers (GPR). The CPU uses rename registers to store the results of an instruction when dispatching the instruction to an execution unit.
Maximum Rename FPRs Reached	FPR	The CPU has used all its available rename floating-point registers (FPR).
VAU Unit Busy	VAUB	The vector arithmetic unit (VAU) is busy executing an instruction.
VAU Unit Not Empty	VAUF	The vector arithmetic unit is not empty
Double Load/Store	DLS	Two load or two store instructions were performed.
LSU Busy	LSUB	The CPU's load/store unit is busy executing an instruction.
Maximum Allowed Dispatch	MAX	The CPU dispatched the maximum number of instructions this cycle.
Mispredict Drain	MD	The CPU is draining a mispredicted branch and any accompanying instructions from the instruction queue.
Tail Serialization	TS	A tail serialization instruction occurred. In a tail serialized instruction, the CPU cannot dispatch new instructions until the serialized instruction finishes.
Maximum Speculation Reached	SPEC	The CPU has executed the maximum amount of speculative instructions.
Fixed Point Unit Busy	FXUB	The CPU's integer unit is busy executing an instruction.
FPU Unit Busy	FPUB	The CPU's floating-point unit is busy executing an instruction.
Maximum Rename VRs Reached	VR	The CPU has used all its available rename vector registers (VR).
VPU Not Empty	VPUF	The vector permute unit (VPU) is not empty.
LSU Full	LSUF	The CPU's load/store unit is full.
System Unit Busy	SUB	The CPU's system register unit is busy.

## CPI

The first piece of data `simg5` shows is the cycles per instruction (CPI) for each interval you specified when you ran `simg5`. If you told `simg5` to run for 500 instructions with a CPI interval of 100, there would be five listings in the output file, one every 100 instructions. The following example shows a typical CPI listing:

```
arch inst 100  time 8560.000000
trace 0 completed arch 100 int 146
CMPL: CPI----- 85.60000
```

The first line of output says `simg5` has executed 100 architected instructions and `simg5` has been running for 8560 clock cycles. The second line says `simg5` executed 100 architected instructions, which expanded to 146 internal instructions. The third line displays the running CPI, which is 85.6 cycles per instruction. Each listing builds on the one before it, which you can see in the listing below.

```
arch inst 201  time 15387.000000
trace 0 completed arch 201 int 309
CMPL: CPI----- 76.55224
```

### Branch Prediction Statistics

As you can tell by the name, the branch prediction statistics section contains statistics that tell you how well the CPU predicted branches in your program. The better the CPU predicts branches, the faster your program runs. When the CPU predicts a branch, it places the first instructions from the predicted path in the instruction stream. If the prediction is correct, the CPU starts executing the instructions it placed in the instruction stream. But if the CPU's prediction is wrong, it must flush those instructions out of the instruction stream and load the instructions from the path your program took, which stalls your program.

`simg5` reports the following branch prediction statistics:

- The accuracy of the branch history table (BHT) at predicting various types of branches.
- A summary of the branch history table's accuracy.
- The count cache statistics. The G5 processor uses a 32-entry count cache to predict the target addresses of branch instructions. The statistics report the percentage of count cache hits and the percentage the count cache was correct.
- The number of branch mispredict flushes.

### Instruction Fetch and Translation Statistics

When the G5 processor fetches an instruction, the first place it looks for the instruction is the effective to real translation table for the instruction cache (IERAT). The effective address is the address you see when you look at your program's memory in a debugger. The real address is the physical memory address. If an instruction isn't in the IERAT, the CPU looks for the instruction in the translation lookaside buffer (TLB). G5 processors have one TLB that contains both instructions and data. If the instruction isn't in the TLB, the CPU looks in the L2 cache for the instruction, then memory.

`simg5` reports the following information:

- IERAT hits and misses.
- TLB hits and misses.
- Instructions found in the L2 cache.
- Instructions found in memory.

## Instruction Cache Statistics

simg5 reports summary information for the L1 instruction cache.

- The probability of an L1 instruction cache hit.
- The probability of an L1 instruction cache miss hitting in the prefetch buffer.
- The probability of an L1 instruction cache miss hitting in the L2 cache.
- The average wait time on an L1 instruction cache miss.

For instruction cache misses and instruction prefetch generation, simg5 reports the number of instances of the instruction being found in the L2 cache and the number of instances found in memory.

## Data Side Translation Statistics

Data translation takes the same steps as instruction translation.

- 1) Look in the effective to real translation table for the data cache (DERAT).
- 2) If the data isn't in the DERAT, look in the translation lookaside buffer (TLB).
- 3) If the data isn't in the TLB, look in the L2 cache.
- 4) If the data isn't in the L2 cache, look in memory.

simg5 reports the following information for data translation:

- The probability of loads and stores being found in the DERAT on the first try.
- The probability of loads and stores being found in the DERAT at any time.
- The probability of data being found in the TLB.
- The number of loads and stores found in the L2 cache.
- The number of loads and stores found in memory.

## Data Prefetch Statistics

If you read the section on simg4, you might remember that prefetching is retrieving data and instructions before they execute so the CPU pipeline stays full. Like G4 processors, G5 processors can prefetch data and instructions. On G5 processors the prefetched data comes from data streams in the AltiVec unit.

simg5 first reports stream summary data. Up to four data streams can be active at one time. simg5 tells you the percentage of time 0, 1, 2, 3, and 4 data streams were active simultaneously.

The prefetch behavior statistics follow the stream summary data. There are too many statistics to cover. Fortunately, simg5 categorizes the statistics to tell you about good and bad behavior. You can look at the bad behavior and discover the problems your program has when prefetching data.

**Data Cache Statistics**

`simg5` reports the following statistics for the L1 data cache:

- The probability of a L1 data cache hit.
- The probability of a L1 data cache miss resulting in a L2 cache hit.
- The average time the CPU waits during load misses and load hit reloads.
- The number of L1 data cache misses found in the L2 cache, separated into loads and prefetches.
- The number of L1 data cache misses found in memory, separated into loads and prefetches.

**Execution Unit Statistics**

`simg5` reports the percentage of cycles an instruction was issued to each execution unit. The G5 processor has the following execution units:

- Two fixed-point (integer) units.
- Two floating-point units.
- Two load/store units.
- Branch register unit.
- Condition register unit.
- Vector permute unit, which handles bit shifts and permute instructions.
- Vector simple integer unit. The simple integer unit can handle only simple integer instructions that execute in one cycle.
- Vector complex integer unit. The complex integer unit handles integer instructions that take longer than one cycle to execute, such as division.
- Vector float unit.
- Vector store unit.

The vector units handle the AltiVec instructions in your code.

**Queue Resource Usage Statistics**

The G5 processor arranges instructions in groups of five before dispatching them for execution. During dispatch the CPU breaks the group into its individual instructions and sends them for execution. When all five instructions are finished, the CPU regroupes them. Grouping instructions allows the G5 processor to manage more instructions simultaneously.

To keep track of all the instructions in the pipeline, the G5 processor has information queues. `simg5` reports the average number of entries in each queue. In addition, `simg5` reports the average number of internal and architected instructions in the pipeline. For more information on internal and architected instructions, refer to the section “Instruction Frequency” later in this chapter.

When the G5 processor dispatches an instruction, the instruction enters an issue queue, which moves the instruction to the execution unit. There are six issue queues.



- The branch issue queue moves instructions to the condition register and the branch register.
- The fixed-point issue queue moves instructions to the integer execution units.
- The floating-point issue queue moves instructions to the floating-point execution units.
- The load/store issue queue moves instructions to the load/store units.
- The vector permute issue queue moves instructions to the vector permute unit.
- The vector shared issue queue moves instructions to the vector store, vector simple, vector complex, and vector float units.

Issue queues maximize the performance of each execution unit. The G5 has two floating-point units, two integer units, and two load/store units. Each of these units has its own issue queue, which determines which of the two execution units executes the instruction. If one execution unit is working on a time consuming instruction, the issue queue moves instructions to the second execution unit until the first unit finishes the time consuming instruction.

`simg5` reports the average number of instructions in each issue queue. The fixed-point (integer) and floating-point execution units have four slots each, with instructions in slots 0 and 3 going to one execution unit and instructions in slots 1 and 2 going to the other unit. `simg5` reports the average number of instructions in each slot.

## Rename Resource Usage Statistics

The G5 processor uses rename registers to store the results of an instruction when dispatching an instruction to an execution unit. Rename registers let you execute instructions out of order; the rename register stores the results until the execution unit needs it. `simg5` reports the average number of rename registers used.

## Instruction Frequency

The instruction frequency section has two tables of information: one for architected instructions and one for internal instructions. Architected instructions, also known as ops, expand into one or more internal instructions, also known as iops. The architected instructions table appears first. For each architected instruction `simg5` tells you the following information:

- The number of instructions.
- The percentage of total instructions.
- The number of internal instructions this instruction expands to.
- The percentage of total internal instructions.
- The ratio of internal to architected instructions.

Let's look at an architected instructions table listing.

```
stw | 1887 | 3.7706 | 3773 | 5.8616 | 1.999 |
```

The listing says the program executed the `stw` instruction 1887 times, which make up 3.7706 percent of the architected instructions. The `stw` instructions expanded to 3773 internal instructions, which make up 5.8616 percent of the internal instructions. The ratio of internal to architected instructions is 1.999.

The internal instructions table follows the architected instructions table. For each internal instruction `simg5` reports the number of instructions and the percentage of total instructions. The report breaks the internal instructions into expanded and non-expanded instructions. For both types of internal instructions, `simg5` reports the number and percentage of instructions. The following example shows an internal instructions table listing:

```
addi | 5752 | 8.9361 | 4311 | 9.6594 | 1441 | 7.3006 |
```

The output says there are 5752 `addi` internal instructions, making up 8.9361 percent of the internal instructions. 4311 of the 5752 instructions were not expanded from architected instructions, making up 9.6594 percent of non-expanded internal instructions. 1441 of the 5752 instructions were expanded from architected instructions, making up 7.3006 percent of the expanded instructions.

## CPI Stack

The CPI stack breaks down the cycles per instruction into categories that `simg5` calls reasons. For each reason `simg5` displays the following information:

- The reason.
- The CPI for the reason.
- The number of cycles.

Let's look at a CPI stack listing.

<code>Wait_for_ifetch</code>	<code>1.40335</code>	<code>70169</code>
------------------------------	----------------------	--------------------

The listing says that `simg5` spent 70169 clock cycles waiting to fetch instructions. Waiting to fetch instructions accounted for 1.40335 CPI.

The report briefly explains each reason. `simg5` breaks down the reasons into the following categories:

- Pipeline contention, which means a group of instructions is waiting to complete and the instructions experienced stall conditions.
- Pipeline empty, which means there are no groups of instructions waiting to complete.
- Pipeline refill, which means a group of instructions is waiting to complete, but there were no stall conditions.

## Pipe Output

Like `simg4`, `simg5` can generate pipe output that lets you look at each instruction your program executed. To generate pipe output, you must run `simg5` with the following options:

```
-p [Scroll Method] -b [Beginning Value] -e [End Value]
```

There are three scroll method values: 1, 2, and 3. Method 1 scrolls by architected instruction count. Method 2 scrolls by internal instruction count. Method 3 scrolls by cycle count. Each architected instruction expands into one or more internal instructions.

*Beginning Value* tells `simg5` when to start scrolling. Normally you want to use the value 1, which tells `simg5` to scroll from the beginning. *End Value* tells `simg5` when to stop scrolling. If you're going to scroll by instruction count, use the number of instructions you're running through `simg5`. If you scroll by cycle count, remember that a trace contains many more cycles than instructions so make sure you supply a large enough value. If you wanted to generate pipe output for the example I used earlier to run `simg5`, you would enter the following command:

```
simg5 thread_001.tt6e 500 25 1 MyTrace -p 1 -b 1 -e 500
```

Running `simg5` for pipe output creates two additional files: one with the extension `.pipe` and one with the extension `.config`. Use the Scroll Pipe Viewer program to see the results. Scroll Pipe Viewer has a GUI, but you must launch it from the command line by typing `scrollpv`.

To look at a pipe file with Scroll Pipe Viewer, choose File > Open > Open File 1. A dialog box opens asking for a pipe file to open. Scroll Pipe Viewer can take several minutes to load the file depending on the number of instructions you traced. On my Mac loading a pipe file with 5000 instructions took over a minute. Scroll Pipe Viewer provides four columns of information for each instruction.

- The instruction number.
- The mnemonic, which is the instruction itself.
- The instruction's memory address.
- The memory address of any data the instruction uses.

The left side of the Scroll Pipe Viewer window contains a grid. Each column in the grid represents a clock cycle, and each row represents an instruction. Initially you see the earliest instructions and cycles. If you scroll down to see later instructions, you have to manually scroll to the right to find the clock cycle where the instruction appears. Manually scrolling gets old quickly. To avoid this annoyance, choose Scroll Mode > Symbol Tracking. A dialog box opens. Select the Track on checkbox. Tell Scroll Pipe Viewer to track on F with an offset of 0. F is the symbol to fetch an instruction. Now when you scroll down to see new instructions, Scroll Pipe Viewer scrolls to the right automatically. The clock cycles where the new instructions were fetched appear in the grid.

The grid on the left side of the Scroll Pipe Viewer window consists of one-character symbols that tell you what an instruction was doing during an individual clock cycle. The grid looks similar to the 30 clock cycle display in `sim4`'s horizontal scroll pipe. Most of the time the symbol is a period, which means the instruction wasn't in the pipeline. Red symbols indicate performance problems, such as stalls and cache misses. Moving the mouse over a symbol fills the control panel above the grid with information about where the instruction was in the instruction pipeline during this clock cycle. As you move left to right on the grid, you can follow the path the instruction takes from fetch to completion.

Clicking a symbol tells you the following information about an instruction:

- The internal instruction ID.
- The architected instruction ID.
- The instruction.
- The instruction's memory address.
- Any read registers and write registers the instruction used.

## acid

`acid` works differently than `sim4` and `sim5`. Rather than simulating how your program runs on a single processor, `acid` focuses on your program's instructions and data accesses. `acid` generates statistics about the instructions and data accesses as well as reporting processor stalls. `acid` reports processor stalls for the PowerPC 7410, 7450, and 970 processors and breaks down the stalls by execution unit.

Running `acid` takes the following form:

```
acid -i [Input File] -o [Output File]
```

*Input File* is the trace file `amber` created for you. *Output File* is where `acid` writes the results. If you don't supply an output file, `acid` writes its results in the shell window.

```
acid -i thread_001.tt6e -o acidResults.txt
```

## Summary Information

When you examine the output file `acid` creates, you'll initially see summary statistics for the instructions and data accesses your program made.

### Instruction Statistics

The first piece of summary information `acid` reports is the number of instructions executed during the trace. `acid` breaks the instructions down by instruction type. For each instruction type `acid` reports the following information:

- The instruction type.
- The number of instructions executed.
- The percentage of total instructions executed.

`acid` breaks down PowerPC assembly language instructions into the following categories:

- Integer, consisting of integer arithmetic, comparison, and bit shifting instructions.
- Floating-point, consisting of floating-point arithmetic, comparison, and conversion instructions.
- AltiVec, consisting of arithmetic and permute instructions in the AltiVec unit.
- Branch, consisting of instructions that change the flow of control in your program.
- Load, consisting of instructions that load data from memory to the registers in the integer, floating-point and AltiVec units.
- Store, consisting of instructions that store data from the registers in the integer, floating-point, and AltiVec units. The CPU stores the data in memory.
- Cache control, consisting of instructions that manage the PowerPC's caches.
- Data stream, consisting of instructions that manage the AltiVec unit's data streams.
- Miscellaneous, consisting of any instructions that don't fit into the other eight categories.

### Use Distance Statistics

`acid` tells you the following use distance statistics: target-use distance, load-use distance, and basic block length. The *target-use distance* measures the number of instructions between an instruction writing to a register and a second instruction using that register. The *load-use distance* measures the number of instructions between an instruction loading data from memory and a second instruction using that data. Shorter use distances are generally better for your program's performance.

The *basic block length* is the number of instructions in a basic block, which is a sequence of instructions that execute in order from beginning to end with no chance of branching. Longer basic blocks execute more efficiently. In the following example:

```
if (x == 0) {
    y = -1 * z;
    z = 1;
}
```

The two statements inside the braces make up a basic block. The basic block length is at least two in this example because there are two statements in the basic block. A statement in a high-level language like C++ breaks down into one or more assembly language instructions. Because `acid` measures basic block length by the number of assembly language instructions, `acid` might report a basic block length greater than the number of high-level language statements in the basic block.

When reporting the target-use distance, the load-use distance, and the basic block length, `acid` reports both the arithmetic mean (a-mean) and harmonic mean (h-mean). The arithmetic mean is what most people think of as the average: take the sum of all the values and divide by the number of values. The harmonic mean is another method of calculating the average. It calculates the average by taking the reciprocal of the arithmetic mean of the reciprocals of the values. The harmonic mean prevents one abnormally large value from having a big impact on the average.

Let's use a simple example to illustrate the difference between the arithmetic and harmonic means. Suppose we want to calculate the average of the values 3, 5, and 10. The arithmetic mean is 6.

$$A = (3 + 5 + 10) / 3 = 6$$

The harmonic mean is approximately 4.739.

$$\begin{aligned} a &= (1/3 + 1/5 + 1/10) / 3 \\ a &= (.333 + .2 + .1) / 3 = .211 \\ H &= 1 / a \\ H &= 1 / .211 = 4.739 \end{aligned}$$

## Branches

`acid` breaks down your program's branches into the following categories:

- Branches taken.
- Branches not taken.
- Backward branches.
- Forward branches. The sum of the forward and backward branches equals the number of branches taken.
- Statically predicted branches.
- Statically mispredicted branches.

## Data Access Statistics

`acid` reports the following information about the data accesses your program makes:

- The number of memory pages your program accessed, separated into instructions and data.
- The number of data accesses your program made.
- The number of load spills. Load spills occur when your program runs out of registers. When the program runs out of registers, the loads spill into memory.

To improve your program's performance, you should reduce the number of memory pages your program uses and reduce load spills. More memory pages means a higher chance of your program running out of memory and having to access the hard disk. Load spills slow your program because retrieving data from memory is slower than retrieving it from a register.

### Stall Cycles

After listing the data access statistics, `acid` reports the number of stalled processor cycles for the PowerPC 7410, 7450, and 970 chips. For each processor `acid` breaks down the stall cycles by execution unit: Load Store Unit, Branch, Integer, Floating Point, and AltiVec.

### Execution Serializing Instructions

`acid` reports the number of execution serializing instructions for the PowerPC 7410, 7450, and 970 processors. Execution serializing instructions do not execute until all prior instructions have completed. Waiting for the prior instructions to complete stalls your program. Reducing the number of execution serializing instructions improves your program's efficiency.

To reduce the number of execution serializing instructions, you need to know what instructions cause execution serialization. Instructions that access or modify non-renamed registers use execution serialization. The following types of instructions are most likely to cause execution serialization:

- Move instructions that move data to and from special purpose registers, such as the link, count, and condition registers.
- Logical operations (AND, OR, NOT, XOR) on the condition register.
- Data stream touch instructions on G5 Macs.

On G4 processors the system register unit (SRU) executes most instructions that use execution serialization. Run `simg4` to see how much your program uses the system register unit.

### Misaligned Accesses

A memory access is aligned if the memory address you access lines up on a boundary equal to the number of bytes you want to access. If you want to access four bytes of memory, the starting memory address must be a multiple of four. The boundary must be a power of two. Common memory alignments are 1 (byte), 2 (halfword), 4 (word), 8 (doubleword), and 16 byte (quadword) boundaries because PowerPC processors normally access 1–16 bytes of memory in a single instruction.

If the boundaries don't line up, the memory access is misaligned. The most common cause of a misaligned memory access is accessing varying sizes of data. Suppose the current address to write to memory is 0004, which is a multiple of four. If you write two bytes of data to memory, the current address becomes 0006. If you proceed to write four bytes of data, the memory access is misaligned because 0006 is not a multiple of four.

Having misaligned memory accesses in your code means your code isn't running as fast as it can. The performance hit for misaligned memory accesses is especially high for floating-point data. A memory access that spans two cache blocks or two memory pages also causes a large slowdown.

`acid` tells you how many misaligned halfword (2 byte), word (4 byte) and doubleword (8 byte) memory accesses your program made.

## Most Used Register List

acid reports the top three integer registers, floating-point registers, and vector registers used in your program. For each register listed, acid places a number in parentheses. The number represents the percentage of the register class instructions where this register appeared. The number for an integer register would show the percentage of integer instructions where this register appeared.

## Detailed Statistics

The detailed statistics section takes the information listed in the summary information section and breaks it down into more detailed information.

## Instruction Mix Statistics

For each instruction type acid shows a breakdown of instruction categories. Integer instructions break down into add, subtract, multiply, divide, comparison, and shift categories. For each category acid reports.

- The number of instructions of this category.
- The percentage of the instruction class this category makes up.
- The percentage of the total instructions this category makes up.

The example below shows an instruction mix listing.

Integer	Add/Sub	203122	37.18	16.52
---------	---------	--------	-------	-------

In this example 203122 integer addition and subtraction instructions executed. The 203122 instructions make up 37.18 percent of the integer instructions executed and 16.52 percent of the total instructions executed.

## Executed Instruction List

In the executed instruction list acid shows each assembly language instruction your program executed. acid sorts the list by instruction count, with the instructions that execute the most appearing first. For each instruction acid shows the following information:

- The name of the instruction.
- Count, the number of times your program executed the instruction.
- Percentage of class.
- Percentage of total.

Let's look at an example of what acid reports in the executed instruction list.

lwz (intload4)	183961	89.80	14.96
----------------	--------	-------	-------

The listing tells us the lwz instruction executed 183961 times. The 183961 lwz instructions make up 89.8 percent of the load instructions executed and 14.96 percent of the total instructions executed.

### Stall Cycles

`acid` reports the stall cycles for each execution unit. The following example shows the stall cycles for the load/store unit:

Stall Cycles Load Store			
	7410	7450	970
1	58948	63152	51954
2	1959	62955	60933
3	0	69	1032
4	1959	0	118
5	0	0	1959

The output says that 58948 instructions stalled the load/store unit one clock cycle on the PowerPC 7410, 63152 instructions on the PowerPC 7450, and 51954 instructions on the PowerPC 970.

### Target-Use Distance Counts

The target-use distance counts break down the writes to registers in your program by target-use distance. The following example shows a piece of `acid`'s target-use distance output:

3	84618
4	48526
5	34600

The output says that 84618 register writes have a target-use distance of three instructions, 48526 register writes have a target-use distance of four instructions, and 34600 register writes have a target-use distance of five instructions.

### Load-Use Distance Counts

The load-use distance counts break down the loads in your program by load-use distance, which you can see in the example below.

11	558
12	593
13	368

The listing says that 558 loads in the program have a load-use distance of 11 instructions, 593 loads have a load-use distance of 12 instructions, and 368 loads have a load-use distance of 13 instructions.



## Basic Block Length Counts

The basic block length counts break down the basic blocks in your program by block length, which you can see in the following example:

1	9691
2	87400
3	76932

The output says the code has 9691 basic blocks one instruction long, 87400 basic blocks two instructions long, and 76932 basic blocks three instructions long.

## Register Use Statistics

For each register `acid` reports the following information about its use:

- The number of instructions that used the register.
- The percentage of instructions of this register class — integer, floating-point or vector — that used the register.
- The percentage of total instructions that used the register.

Let's look at an example of `acid`'s register use statistics.

R0	358200	19.48	19.22
----	--------	-------	-------

The listing says 358200 instructions used the integer register R0. The 358200 instructions make up 19.48 percent of the instructions that used an integer register and 19.22 percent of the total instructions.

## Data Address Statistics

`acid` lists the address of every memory page where your program accessed data along with the number of times your program accessed the page. First, `acid` sorts the listings by memory page, with the lowest memory addresses appearing first. Later in the report, `acid` sorts the listings by usage, with the memory pages accessed the most appearing first.

## Instruction Address Statistics

`acid` lists the address of every memory page where your program executed an instruction along with the number of times your program accessed the page. First, `acid` sorts the listings by memory page, with the lowest memory addresses appearing first. Later in the report, `acid` sorts the listings by usage, with the memory pages accessed the most appearing first.

## Gathering Data on Each Instruction

If the information `acid` generates in its standard report isn't enough for you, `acid` has two options that list each instruction your program called. The list of instructions appears along with the rest of the information `acid` reports.

### **-a Option**

The `-a` option lists the following information for each assembly language instruction your program executes:

- Instruction number.
- Program counter address.
- Stalls, which consists of three numbers in brackets. The first number represents the number of cycles the instruction would stall on the PowerPC 7410 processor. The second number represents the number of cycles the instruction would stall on the PowerPC 7450. The third number represents the number of cycles the instruction would stall on the PowerPC 970.
- The instruction.
- Any performance problems like stalls, mispredicted branches, serializing instructions, and load spills.

If you trace an entire program, you should be aware that even small programs can execute hundreds of thousands of instructions. Listing hundreds of thousands of instructions results in output files that can approach 100 MB in size. You should use the `-a` option when tracing a single function.

### **-A Option**

The `-A` option lists all the information for each instruction the `-a` option does. `acid` lists the following additional information for each instruction when you run it with the `-A` option:

- Branch PC, the branch target address. Only branch instructions have any data in the Branch PC field.
- Data EA, the data address for load and store instructions.
- Third word, which consists of any extra information in the instruction. The third word is usually empty.
- TU, the target-use distance.
- Up, the target-use distance for the update register, which updates load and store instructions.
- CR, the target-use distance for the condition register. Only instructions that set the condition register have data in the CR column.
- LK, the target-use distance for the link register. Only instructions that set the link register have data in the LK column.

# Chapter 7

## MallocDebug

MallocDebug is a program to help you understand how your program uses memory. It tells you where your program allocates memory and how much memory your program uses. In addition, MallocDebug alerts you to memory-related problems in your code, including memory leaks and memory overruns.

### Running MallocDebug

After launching MallocDebug you have two options: launch your application from MallocDebug by choosing File > New Window or attach your running application to MallocDebug by choosing File > Attach. I strongly recommend launching your application from MallocDebug. To attach a running application to MallocDebug, the application must be linked to the MallocDebug library. When you launch your program from MallocDebug, MallocDebug dynamically links the MallocDebug library to your program. If your application is currently running, quitting it is a lot faster than going into Xcode, adding the MallocDebug library to your project, and recompiling the program.

#### NOTE

You can also use Xcode to run your program in MallocDebug. Open your project in Xcode and choose Debug > Launch Using Performance Tool > MallocDebug. The Executable field in the MallocDebug window will contain the path to your program.

The MallocDebug window, which you can see in Figure 7.1, appears after you decide how to run your program in MallocDebug. If you're running MallocDebug for the first time, use the Browse button. Clicking the Browse button brings up an Open File dialog box; use it to select your application. The Executable combo box contains a list of the most recent programs on which you've run MallocDebug. Use the list to select your application if you've run it through MallocDebug recently. The Arguments text field allows you to specify any runtime arguments your application needs to run. Most Mac programs don't use runtime arguments.

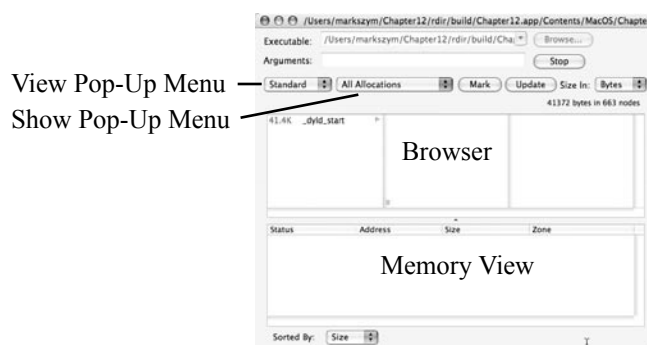


Figure 7.1

MallocDebug window.

Click the Launch button to launch your program. The Launch button will change to a Stop button. Don't click the Stop button until you're finished examining your program with MallocDebug. Clicking the Stop button quits your program, preventing MallocDebug from showing any information about the program. Run your application the way a typical user would. Click the Update button to see the results.

## Examining Your Program's Memory Usage

Above the browser on the left side of the window are two unlabeled pop-up menus. The left menu, which I will refer to in this chapter as the View pop-up menu, tells MallocDebug how to display your program's functions in the browser. The initial choice is Standard, which means MallocDebug starts displaying functions at the start of the call tree. The right menu, which I will refer to in this chapter as the Show pop-up menu, tells MallocDebug what to display in the browser. The initial choice is All Allocations, which means MallocDebug displays information on all the memory allocations your program makes.

Assuming you don't change the two unlabeled pop-up menus, the first piece of information MallocDebug reports when you click the Update button is the total amount of memory your program allocated. MallocDebug displays the amount of memory allocated right above the browser. The more memory your program uses, the greater the chance of the operating system having to go to disk to free up more memory, which slows down your program.

In the standard view the left pane of the browser contains the first function your program called. Next to each function in the MallocDebug browser is a number. By default the number is the number of bytes your application allocates while this function appears in the call stack. Choosing Count from the Size In pop-up menu tells MallocDebug to display the number of memory allocations next to each function. The first function your program calls stays on the call stack until you quit the program, which is why the number of bytes next to the first function equals the number of bytes your program allocated.

Should you discover your program allocates too much memory, you want to find how much memory each function allocates so you can find ways to use less memory. Use the browser to navigate the call tree. If a function in the browser has a triangle next to it, the function calls additional routines. Select the function to see the routines that function calls. In the standard view moving left to right in the browser moves you down the call tree. When you select a function from the browser, MallocDebug displays the following information above the browser:

- The function's name.
- The amount of memory allocated while this function appears in the call stack.
- The number of nodes, which is the number of memory allocations made while this function appears in the call stack.

When examining memory usage, what's most difficult to determine is how much memory a function in your code allocates. The first step to determining a function's memory usage is to find a function that allocates memory. If a function you select in the browser allocates memory, the memory view shows each allocation. Navigate the call tree in the browser until you find a function that allocates memory.

Most functions that actually allocate memory are functions you did not write. If you call a Carbon function to create a window, the Carbon function calls a series of low-level functions, one of which allocates the memory for the window. The second step to determining a function's memory usage is to backtrack from the memory allocating function to a function you wrote. The backtracking tells you the function in your code responsible for the memory allocation. Selecting the responsible function in the browser tells you how much memory this function allocates.

Functions you wrote have an icon next to them in the browser. C, C++, and Objective C functions have an icon of a sheet of paper with the letter C. Java functions have the letter J in the icon instead of C. Double-clicking a function with an icon opens the function's source code file in Xcode and takes you to that function in Xcode. Xcode must be running to be able to open the file.

## Flat View

If you choose Flat from the View pop-up menu, MallocDebug flattens the call tree. The flat view places all the functions your application calls in the left pane of the browser. If multiple routines call a certain function, that function appears once in the flat view. By using the flat view you can look at a function and see the total amount of memory allocated or the number of memory allocations made while this function appeared in the call stack.

## Inverted View

If you choose Inverted from the View pop-up menu, MallocDebug inverts the call tree. The browser starts at the bottom of the call tree with the leaf functions. Moving left to right in the browser moves you up the call tree. Inverting the call tree helps when you want to look at individual memory allocations because the allocations tend to occur at the bottom of the call tree. Inverting the tree makes the memory allocations appear more quickly in the browser.

## Examining Individual Memory Allocations

When a function allocates memory, the memory view contains a list of each allocation the function made, as you can see in Figure 7.2. The memory view contains four pieces of information about each memory allocation: Status, Address, Size, and Zone. The Status field is normally blank; it contains information only if you have a memory overrun or underrun in your program. The Address field contains the starting memory address for this allocation. The Size field contains the size (in bytes) of the allocation, and the Zone field shows the memory zone where this allocation occurred. The most common zone is `DefaultMallocZone`.

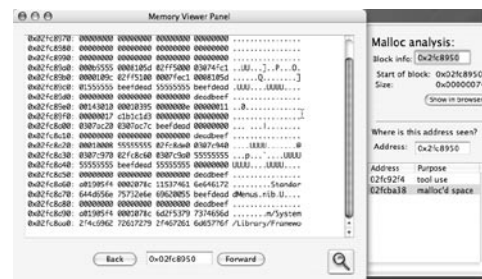
By default MallocDebug sorts the allocations by size, with the largest allocations appearing first. The Sorted By pop-up menu below the memory view gives you two additional options: Time and Zone. The Time option sorts the allocations by the time they occurred with the earliest allocations appearing first. The Zone option sorts by memory zone.

Double-clicking an entry in the memory view opens the memory viewer panel, shown in Figure 7.3. The memory viewer panel lets you look at the memory where the allocation occurred, which probably comes as a shock to you. The Back and Forward buttons function the way the Page Up and Page Down keys do when reading a document on your computer.

Status	Address	Size	Zone
	0x5130000	3150808	DefaultMtlLocZone
	0x5135000	2895244	DefaultMtlLocZone
	0x5641000	4666528	DefaultMtlLocZone
	0x5646000	4666444	DefaultMtlLocZone
	0x6070000	1331872	DefaultMtlLocZone
	0x6061000	97664	DefaultMtlLocZone
	0x6063000	97664	DefaultMtlLocZone
	0x5634000	16900	DefaultMtlLocZone
	0x5630000	16900	DefaultMtlLocZone

### Figure 7.2

Memory view.



### Figure 7.3

Memory viewer panel.

When looking at the contents of memory, you may see the values `beefdead` and `deadbeef`. MallocDebug places the word `beefdead` before a block of memory and `deadbeef` after the block to guard the block. MallocDebug uses the `beefdead` and `deadbeef` words to detect memory overruns and underruns. Another memory value you may see is `55555555`. When your program frees a block of memory, MallocDebug fills the block with the value `55555555`. If your program tries to access memory with the value `55555555`, MallocDebug crashes your program, alerting you to a bug in your program.

Clicking the button with the magnifying glass icon opens the Malloc Analysis drawer. Enter a memory address in the Block Info text field. The address you enter must be the start of a block of allocated memory in your program. MallocDebug displays the size of the memory block. It also reports the address of the start of the memory block, which isn't helpful because the address you enter in the text field must be the start of a memory block. Entering a memory address in the Address text field creates a list of the memory addresses that contain the memory address you entered.

## Finding Memory Leaks

A *memory leak* occurs in your program when you allocate memory and forget to free it. The leaked memory cannot be used by the operating system. One small leak won't cause any problems, but large or frequent leaks are big trouble. Suppose your application leaks 100KB of memory every time the user opens a window. If the user opens enough windows, the computer will eventually run out of memory, crashing your program.

Memory leaks are difficult to find by looking through your source code, but MallocDebug makes finding memory leaks easy. Choosing Leaks from the Show pop-up menu tells MallocDebug to display memory leak information in the browser instead of memory allocation information. The number next to a function in the browser represents either the number of memory leaks or the amount of leaked memory when this function appears in the call stack. The Size In pop-up menu determines what the number represents.

Discovering whether or not you have memory leaks in your program requires no effort. If you have memory leaks, you want to know where you're leaking memory so you can fix the leaks. Locating the source of your memory leak requires more work. Finding the location of your memory leaks is much easier if you invert the call tree. Inverting the call tree — choose Inverted from the View pop-up menu — brings the functions leaking memory to the front. Work your way up the call tree to find the function in your code responsible for the memory leak.

## Finding Memory Overruns and Underruns

A *memory overrun* occurs when your program writes past the end of a block of memory. A *memory underrun* occurs when your program starts writing before the start of a block of memory. Memory overruns occur more often than underruns. The most common cause of memory overruns is writing past the end of an array. Arrays in the C programming language start at index 0; an array with 100 elements has indices 0-99. If you start writing to the array at index 1 and have one write for each element of the array, the last write causes an overrun.

To look for memory overruns and overruns, choose Over/underruns from the Show pop-up menu. The MallocDebug browser shows the memory overruns and underruns in your program. The number next to each function in the browser represents either the number of overruns and underruns or the amount of overrun or underrun memory. The Size In pop-up menu determines what the number represents.

If you have memory overruns or underruns in your program, you want to know where the overruns and underruns appear in your code so you can fix them. Inverting the call tree — choose Inverted from the View pop-up menu — brings the functions with the overruns and underruns to the front. Work your way up the call tree to find the function in your code responsible for the overrun or underrun.

The MallocDebug browser tells you about the overruns and underruns in your program, but it doesn't tell you whether you have an overrun or underrun. To see whether you have overruns or underruns, navigate the call stack until the memory view fills with information. In the memory view the Status column reports < for a memory underrun and > for a memory overrun.

## Inspecting Memory Zones

Choosing Tools > Zone Inspector opens the zone inspector panel, which lets you look at the memory zones your program has. The zone inspector panel lists each memory zone along with the amount of memory allocated in each zone. Choosing Counts from the pop-up menu displays the number of allocations made in each zone instead of the amount of memory allocated. Selecting a zone and clicking the Forget Zone button removes the zone from the panel. Be careful when forgetting zones. Once you forget a zone, you can't bring it back.

For the zone inspector to help you, your program must have multiple memory zones. Many Mac programs have only one memory zone, DefaultMallocZone, which makes the zone inspector meaningless. Cocoa programs are the most likely to use multiple memory zones, but not all Cocoa programs have multiple memory zones. Don't worry if your program has only one memory zone. You can use mappings, which I discuss in the section "Organizing Memory Allocations with Mappings" later in this chapter, to simulate memory zones.

## Narrowing Your View

The call trees for all but the most trivial programs can become large, making call tree navigation tedious. Large call trees also make discovering the amount of memory your functions allocate difficult. MallocDebug gives you several ways to reduce the number of functions shown in the browser.

### Viewing Recent Memory Usage

After looking at your program's memory usage, you may want to resume running your application and examine the new allocations your program made. To look at the new memory allocations:

- 1) Click the Mark button.
- 2) Choose Allocations from mark from the Show pop-up menu.
- 3) Click the Update button.

After clicking the Update button, the browser shows only the allocations your program made since clicking the Mark button, the new memory allocations.

### Pruning Functions from the Call Tree

Pruning temporarily removes functions from the call tree, hiding functions you don't care about. Hiding functions you don't care about lets you focus on the important areas of your code. When you prune a function, the memory it allocates goes away as well, which means MallocDebug shows your program allocating less memory when you prune functions.

Use the Prune menu to do your pruning. There are four ways to prune in MallocDebug.

- Prune selection removes the function you select from the call tree. For pruning a selection to have any effect, the function you prune must allocate memory. Switching to the inverted view makes pruning selections easier.
- Prune path removes the function you select and all the functions beneath it in the call tree.
- Prune zone removes any functions that allocate memory in a particular memory zone. Your program must have multiple memory zones to be able to prune a zone.
- Prune not in this library removes functions that are not in the same library as the selected function, allowing you to focus on memory allocations made in a specific library.

At the start of this section, I said pruning temporarily removes functions from the call tree. To restore pruned functions to the call tree, choose Prune > Unprune. Unpruning restores the call tree to its original state. To restore some of the pruned functions, open the filter panel by choosing Tools > Filter > Filter Panel. The filter panel contains the list of pruned functions. Select a function and click the Restore button to add it to the call tree.

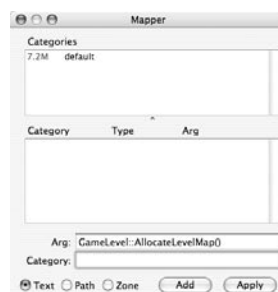
## Organizing Memory Allocations with Mappings

Mappings let you organize your program's memory allocations into categories that you create. If you were writing a word processor, you could have separate mappings for launching the program, creating a new document, spell checking, and printing. With mappings you can organize memory allocations according to your program's features so you can determine how much memory your program allocates when performing each feature. You can map a memory allocation into a category in three ways.

- Mapping by text lets you add individual functions to a category.
- Mapping by path lets you add a group of functions to a category. When you select a function and add its path to a category, MallocDebug adds the selected function and the functions beneath it in the call tree to the category.
- Mapping by zone lets you add an entire memory zone to a category. Your program must have multiple memory zones to make mapping by zone useful.

To create categories and mappings, choose Tools > Mappers > Mapper Panel to open the mapper panel, which you can see in Figure 7.4. Use the radio buttons at the bottom of the window to choose the way you want to map. Enter the function name or zone name in the Arg text field. Enter the name of the category in the Category text field. You can give a category any name you want. Click the Add button to add the function or zone to the category. After creating the mappings, click the Apply button to take MallocDebug's memory information and break it down into the categories you specified.

Typing function names can be tedious. To avoid typing function names, open the mapper panel. Select the function you want to add from the MallocDebug browser. Choose Tools > Mappers > Enter > Text if mapping by text, or choose Tools > Mappers > Enter > Path if mapping by path. The name of the function appears in the Arg text field, ready for you to add.



**Figure 7.4**

Mapping panel.



# Chapter 8

## ObjectAlloc

ObjectAlloc is a tool to examine your program's memory allocations. ObjectAlloc records every memory allocation your program makes, making it the perfect complement to MallocDebug. MallocDebug focuses on your program, telling you how much memory you're allocating and where you're allocating it. ObjectAlloc focuses on the memory allocations. With ObjectAlloc you can look at each memory allocation in your program and see the size of the allocation, the contents of memory, and the function in your program that made the allocation.

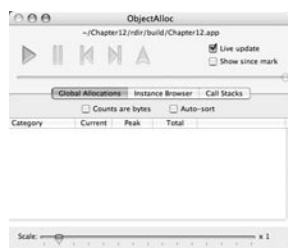
### Running ObjectAlloc

When you launch ObjectAlloc, an Open File dialog box appears for you to choose an application to test with ObjectAlloc. After selecting an application, an ObjectAlloc window like Figure 8.1 opens. ObjectAlloc has five navigation buttons at the top of the window.

- Start/Stop Task button, where the task is your program.
- Pause/Resume Task button.
- Step Backward button.
- Step Forward button.
- Set Mark button.

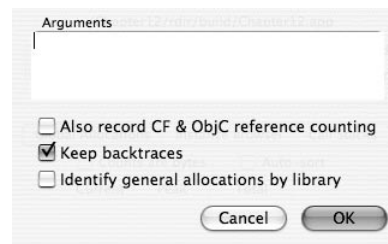
Click the Start Task button. A dialog box opens, as shown in Figure 8.2. In Figure 8.2, you can see there are three checkboxes you can select when running your application with ObjectAlloc.

- The Also Record CF and Obj-C reference counting checkbox will be interesting to you if you write Cocoa programs in Objective C. Objective C objects use a retain count for memory management. When you create an object, the computer allocates memory for the object and sets the retain count to one. If another object wants to use the newly created object, you send a retain message to this object, incrementing the retain count. When you don't need an object any more, send the object a release message, which decrements the retain count. When the retain count reaches zero, the computer frees the memory for the object. If you select the Also Record CF and Obj-C reference counting checkbox, ObjectAlloc records all the retain and release messages your application sends.



**Figure 8.1**

ObjectAlloc window..



**Figure 8.2**

Dialog box to launch a program with ObjectAlloc.

- Selecting the Keep backtraces checkbox tells ObjectAlloc to record the call stack for each memory allocation. Recording the call stack lets you find the functions in your code that are allocating memory. If you want to use the Event Inspector window and use the Call Stacks tab in the ObjectAlloc window, make sure you select the Keep backtraces checkbox.
- Selecting the Identify general allocations by library checkbox tells you which library made a particular allocation. If you do not select this checkbox, most of your memory allocations appear as GeneralBlock. This option is useful to narrow down where an allocation occurred.

After clicking the OK button in the dialog box, ObjectAlloc launches your application. If you selected the Live update checkbox in the ObjectAlloc window, the window immediately starts displaying your program's memory allocations. If the Live update checkbox isn't selected, you must click the Pause Task button to see the memory allocations. Live updating provides instantaneous feedback, but makes your program run slower.

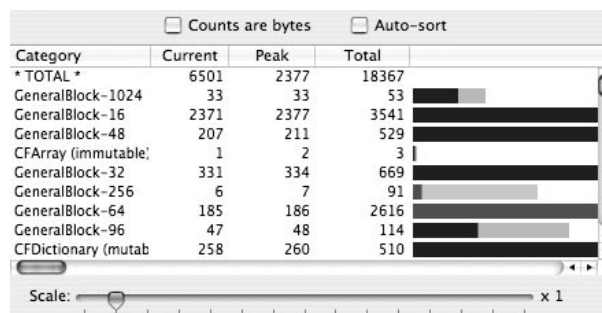
## NOTE

You can also use Xcode to run your program in ObjectAlloc. Open your project in Xcode and choose Debug > Launch using Performance Tool > ObjectAlloc. ObjectAlloc launches, and the ObjectAlloc window shown in Figure 8.1 appears.

## Global Allocations View

The default view in an ObjectAlloc window is the global allocations view. Figure 8.3 shows the ObjectAlloc window with the global allocations view. The global allocations view gives you four pieces of information about the memory allocations your program makes: category, current allocations, peak allocations, and total allocations.

There's one listing for each memory allocation category. Categories take one of two forms. The first form involves code libraries. Categories involving libraries have a number next to them. The number represents the number of bytes of memory the computer allocated in a single memory allocation. If a library makes five 100-byte allocations, three 500-byte allocations, and seven 1000-byte allocations, that library has three categories: one for the 100-byte allocations, one for the 500-byte allocations, and one for the 1000-byte allocations. The library name ObjectAlloc displays depends on whether or not you selected the Identify general allocations by library checkbox when you launched your application. If you did not select it, the library name is GeneralBlock. If you did select the checkbox, ObjectAlloc displays the library name.



**Figure 8.3**

Global allocations view.

The second form of memory allocation involves low-level operating system data structures. Common data structure categories include.

- Core Foundation data types, which have the prefix CF.
- Cocoa classes, which have the prefix NS.
- CoreGraphics data types, which have the prefix CG.

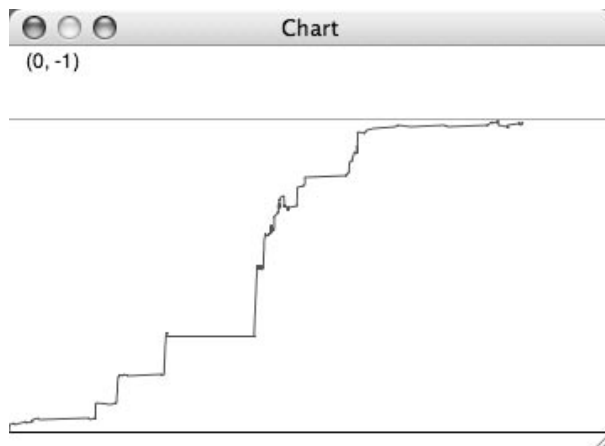
By default the Current column tells you the number of current memory allocations the computer has made. A current allocation is a memory allocation your program makes that it hasn't freed. If you select the Counts are bytes checkbox at the bottom of the window, the Current column tells you the number of bytes of memory the computer allocated. The Peak column tells you the highest number of current allocations the computer has made since you started running your application. The Total column tells you the total number of memory allocations the computer made.

Next to the total value for each listing is a graph. The graphs come in three colors: blue, yellow (It may look orange on your monitor), and red. Most of the time the graph's color is blue. The graph is yellow when the peak number is less than one third of the total number. The graph is red when the peak number is less than one tenth of the total number.

Each of the three graph colors has three shades: dark, medium, and light. The current allocations appear in the dark shade. The peak allocations appear in the medium shade. If the current and peak values are close, seeing the peak allocations can be difficult. The rest of the memory allocations appear in the light shade. The Scale slider lets you specify how many allocations (or bytes if you selected the Counts are Bytes checkbox) one line in the graph represents. The range is .5 to 8192. Use a lower scale for individual allocations and a higher scale for bytes.

Clicking one of the column headings sorts the listing according to the column you choose. Sorting by category sorts the listings alphabetically by category name. Sorting by Current, Peak, or Total sorts the listings by the number of allocations with the highest numbers appearing first. If you select the Counts are bytes checkbox, ObjectAlloc sorts the listings by bytes of memory allocated with the highest numbers appearing first.

Double-clicking a listing in the global allocation view opens a chart for the listing, which you can see in Figure 8.4. There are two horizontal lines running across the chart. The black line runs across the bottom of the chart, and it represents the number zero. The red line represents the peak number of allocations for this listing. The chart contains plots of the current allocation value with one plot per memory event. Initially the upper left corner of the chart has the value (0, -1), which means you haven't selected a plot. Clicking on the graph places a blue line where you clicked, and the number in the upper left corner states the event number. Selecting an area of the graph creates a blue rectangle. The left value is the left edge of the rectangle and the right value is the right edge of the rectangle.



**Figure 8.4**

Chart of current allocation values.

## Instance Browser View

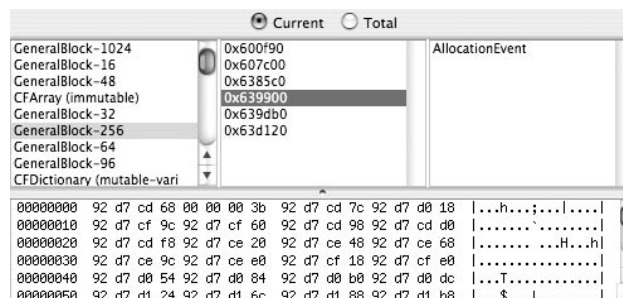
The global allocations view provides an overview of the memory allocations your application makes. To investigate individual allocations, you must click the Instance Browser tab to change the view in the ObjectAlloc window to the instance browser view, which you can see in Figure 8.5.

The instance browser view has three panes that take up the middle third of the window. The left pane contains the list of allocation categories sorted the way you had them sorted in the global allocations view. Selecting one of the categories fills the center pane with the addresses of each memory allocation in the category. Selecting one of these addresses fills the bottom of the window with information about that memory allocation unless the memory has already been freed. If the memory has been freed, the bottom of the window says <freed value>.

In most cases, the bottom portion of the ObjectAlloc window shows the contents of the memory at the address you selected, which Figure 8.5 shows. The amount of memory shown depends on the size of the allocation; a 150-byte allocation means 150 bytes of memory appears in the bottom of the window. The numbers on the left side (the 8-digit hexadecimal numbers) are the offset from the memory address you selected. If the memory address you selected was 0x10123456, the first row would show the contents of memory location 0x10123456, the second row would show 0x10123466, the third row 0x10123476, and so on.

Many Core Foundation data type (allocations whose library name starts with CF) and Cocoa class (allocations whose library name starts with NS) allocations provide better information about the memory allocation. The exact details depend on the particular class. A `CFArray` allocation (a Core Foundation array) shows the array's contents when you select the memory address in the middle pane. An `NSMenu` allocation tells you the menu's title and its menu items when you select its memory address.

I haven't forgotten the right pane. Selecting a memory address from the center pane fills the right pane of the instance browser view with the memory events for the selected memory allocation. Every allocation has an `AllocationEvent` listing, which occurs when you allocate the memory. If your program freed the memory, a `FreeEvent` listing appears in the right pane as well. If you selected the Also record CF and ObjC reference counting checkbox when you launched your application, you will get retain and release events in the Core Foundation data structures and Cocoa classes. In the Core Foundation data structures, these events appear as `CFRetainEvent` and `CFReleaseEvent`. In the Cocoa classes, these events appear as `ObjectRetainedEvent` and `ObjectReleasedEvent`.



**Figure 8.5**

Instance browser view.

Selecting a memory event opens the Event Inspector window, shown in Figure 8.6. The event inspector window provides you with a great deal of detail about the event, including.

- The event number.
- The event type.
- The time index, when the event occurred.
- The size of the memory allocation.
- The pointer, the memory address where the event occurred.
- Any extra data about the event.

The event inspector window also provides a backtrace of the call stack when the event happened if you selected the Keep backtraces checkbox when launching your application in ObjectAlloc. If a function you wrote appears in the call stack, a link to the name of the file and line number appears next to the function. Clicking the link opens the file in Xcode, but you must be running Xcode or else clicking the link does nothing.

## Call Stacks View

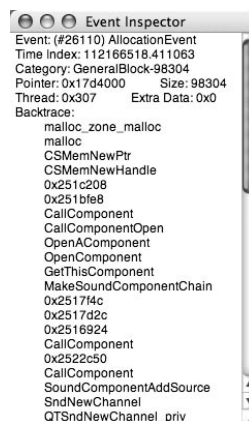
Clicking the Call Stacks Tab changes the ObjectAlloc window to the call stack view. The call stack view provides a way to find where groups of memory allocations occur in your code. Each listing in the call stack view has four columns.

- The memory allocation category.
- The count, which is the number of memory allocations.
- The size, which is the total number of bytes allocated.
- The last column is a function in the call stack. The initial function name is `start()`, which is the first function Mac OS X applications call when the user launches them.

You can sort the listings by category, allocation count, or allocation size. Click the appropriate column.

If you selected the Keep backtraces checkbox when you launched your application with ObjectAlloc, the `start()` function has a disclosure triangle next to it. Clicking the disclosure triangle expands the call stack, letting you see the routines this function called. Keep clicking disclosure triangles to navigate the call stack. Expanding the call stack creates listings in the call stack view with each new listing having a count and size.

If a function in the call stack is a function you wrote, a link with the name of the file and line number appears next to the function. Clicking the link opens the file in Xcode, assuming Xcode is currently running.



**Figure 8.6**

Event inspector window.

## Stepping Through Allocations

Because ObjectAlloc stores every memory allocation your program makes, you can examine any memory allocation. You can even go to the start of your program and examine every memory allocation in sequence. In most cases you won't want to go through the trouble of looking at every memory allocation. Small programs have thousands of allocations with larger programs having tens and even hundreds of thousands of allocations. Normally you will examine only certain sequences of memory allocations.

If you're going to step through a series of memory allocations, there are two things you can do to make the stepping go more smoothly.

- Pause your program in ObjectAlloc. If you leave your program running, memory allocation events continue to occur, which can be distracting if you turned on live updating.
- Open the event inspector window by choosing File > Inspector. Opening the inspector window lets you see all the relevant information about a memory allocation.

To pick a starting point for your stepping, move the unlabeled slider located in the upper third of the ObjectAlloc window. Notice how the contents of the event inspector window change to reflect the moving of the slider. In the global allocations view you can see that the Current, Peak, and Total values match where you are in the memory allocation history.

Click the Step Forward button to move to the next memory allocation. Clicking the Step Backward button moves you to the previous allocation. Dragging the slider allows you to jump around so you don't have to sequentially walk through thousands of memory events.

## Finding a Specific Memory Event

Stepping through individual memory allocations becomes tedious due to the sheer number of memory-related events Mac OS X programs generate. Usually you're interested in only certain types of events. ObjectAlloc lets you search for the events you're interested in.

For event searching to be effective, you should open the event inspector window. Choose Edit > Find > Find Event to open the Find Event dialog box, which you can see in Figure 8.7. You can search for an event based on any of the information the event inspector window displays except for the event's time index. The most interesting criteria is the backtrace. Supply a function from your code and click the Next button to tell ObjectAlloc to find the next event in which the function appears in the call stack. If you have the event inspector window open, it shows the information about the event ObjectAlloc found.



**Figure 8.7**

Find Event dialog box.

You can combine search criteria to refine the search. Supplying the category name `CFArray` and size 64 limits the search to `CFArray` memory events 64 bytes in size.

## Showing Fresh Memory Allocations

After examining memory allocations with `ObjectAlloc`, you may want to run your program some more. When you go back to `ObjectAlloc`, you want to focus on the memory allocations made during the most recent run through your program. `ObjectAlloc` lets you separate the new memory allocations from the old ones.

Move the unlabeled slider to the spot where you want `ObjectAlloc` to start showing memory allocations. Click the Set Mark button to set a mark at that spot. Setting a mark doesn't change `ObjectAlloc`'s output unless you select the Show since mark checkbox. By selecting the Show since mark checkbox, you tell `ObjectAlloc` to display only the memory allocations made after the mark you set.

To see new memory allocations, set a mark at the right end of the slider. Run your program some more, and go back to `ObjectAlloc`. If you select the Show since mark checkbox, only the new allocations appear.

# Chapter 9

## Command-Line Debugging Tools

Over the years Unix programmers have created many command-line tools to help them find problems with their code. Because Mac OS X has Unix at its core, you have access to all these tools as well. I do not recommend starting with the command-line tools. You will find it much easier initially to use Xcode's debugger (covered in Chapter 2), MallocDebug (covered in Chapter 7) and ObjectAlloc (covered in Chapter 8). When you need to dig deeper to find a nasty bug, try the tools in this chapter.

### A Command Line Primer

For those of you who have used only graphical user interfaces, the Unix command line in Mac OS X may seem intimidating, but this section introduces you to enough of the Unix environment for you to use the tools covered in this chapter. To reach the command line, run the Terminal application. It should be in your Applications folder under Utilities. After launching Terminal a Unix shell window opens, and you can start working in the command line.

### Executing Commands as root

Several of the tools I cover in this chapter require you to run them as the `root` superuser, but you don't have to switch to `root`. The `sudo` command allows you to execute any Unix command as though you were `root`. Supply the command you want to execute or the program you want to run.

```
sudo Command
```

After running the `sudo` command you will be prompted for a password. Enter the password for your user name. The command executes after you enter your password.

### Navigating Directories

If your Mac is like mine, it has thousands of directories (folders) for applications, source code, software development kits, and programming documentation. Navigating these directories is more difficult to do from the command line than from the Finder. Two UNIX commands help you navigate your computer's directories: `ls` and `cd`.

The `ls` command lists the files and subdirectories inside the current directory. Without the `ls` command you would have to memorize the contents of your computer's folders to do any navigation.

The `cd` command changes the current directory. To change to a subdirectory of the current directory, enter the subdirectory name. Suppose the current directory has a subdirectory `Games` that holds your favorite games. If you wanted to move to the `Games` directory, you would type.

```
cd Games
```



To move through several levels of subdirectories at once, use slashes to separate the subdirectories. Suppose you wanted to move to the directory where you have the game Halo. You could execute the `cd` command twice, once to move to the `Games` directory and once to move to the `Halo` directory. Or you could move directly to the `Halo` directory with one `cd` command.

```
cd Games/Halo
```

To move back one directory, use `..` as the directory name. If you were in the `Halo` directory and wanted to move back to the `Games` directory, you would type.

```
cd ..
```

If you have a leading slash in the directory name, the directory moves relative to the hard disk on which you installed Mac OS X. To move to your `Developer` directory where the Xcode tools reside, you would type.

```
cd /Developer
```

If one of your directories has a space in it, put the directory name in quotes.

```
cd "Source Code"
```

Without the quotes, the computer treats the directory name `Source Code` as two separate arguments, and you will get an error message "Too many arguments".

## Getting Help

If you get stuck in the command line, Unix manual pages are your friends. There are manual pages for virtually every Unix system command and command-line tool. They describe what the command (or tool) does, the arguments it takes, and the available options. To read a manual page, use the `man` command.

```
man CommandName
```

To learn about the options available for the `ls` command, type.

```
man ls
```

Most manual pages are too large to fit in a shell window so only the first part of the manual page appears in your window. The bottom line of your window will have a colon with the cursor next to it to tell you there's more text to read. To see the next line of text, press the down arrow key or the Return key. To see the next page (the amount of text that will fit in the window) of text, press the space bar. Press the up arrow key to see previously viewed parts of the manual page that are no longer visible in the window.

### NOTE

You can read Unix manual pages in Xcode. Choose `Help > Open man page`.

## Finding Your Application's Process ID

Several of the tools in this chapter require you to supply your application's process ID. Every running application in Mac OS X has a process ID, which is a number that uniquely identifies the application to the operating system. You have no way of knowing your application's process ID until you launch it.

To find your program's process ID, open a new shell window by choosing File > New Shell. Run the `top` tool by typing the word `top`. `top` lists all the currently running processes with information about each process, such as how much CPU time and memory they are using. When you run `top`, look at the left side of the shell window to see a column with the heading `PID`. This column lists each running application's process ID. Find your application's process ID and be ready to use it when running the debugging tools in this chapter.

## `fs_usage`

The `fs_usage` tool presents a real-time listing of file system calls and page faults. Before running `fs_usage`, make your shell window as wide as possible; the wider the window, the more information `fs_usage` returns. You must use the `sudo` command to run `fs_usage`.

## Reporting File Manager Routines

Cocoa and Carbon programs use the File Manager to work with files. The File Manager functions call the low-level file system routines that `fs_usage` reports. If you use Cocoa or Carbon in your program, you want `fs_usage` to report the File Manager function calls as well as the low-level routines. Use the debug runtime library in your program to display the File Manager functions your program calls. To tell Xcode to use the debug runtime library, run Xcode and open your project.

- 1) Select Executables from the Groups and Files list. The list of executable files appears in the window.
- 2) Select the executable name of the program you want to profile.
- 3) Click the Info button to open the information panel for the executable file.
- 4) Click the General tab.
- 5) You should see a pop-up menu with the label Use suffix when loading frameworks (See Figure 5.2 in Chapter 5). Choose debug from the menu.
- 6) Clean your project by choosing Build > Clean.
- 7) Build your project by choosing Build > Build.

## Running fs\_usage

If you supply no arguments to `fs_usage`.

```
fs_usage
```

It displays information on all running processes. Running `fs_usage` on all running processes is usually a bad idea. `fs_usage` displays each file system call as it occurs, and Mac OS X applications make a surprisingly large number of file system calls, making it hard to keep up with all the information `fs_usage` reports. In most cases the only process you're interested in is your program. To limit `fs_usage`'s reporting to your application, supply either the application's process ID or its name.

```
fs_usage ProcessID
```

```
fs_usage AppTitle
```

When you run `fs_usage` on your application, the shell window seems to lock up with no data appearing in it. The lack of activity reflects the fact that you're in the Terminal application, which means your application isn't doing anything at the moment. Switch back to your application, do some things in it, then switch back to Terminal. Your shell window should be packed with data from `fs_usage`.

## What fs\_usage Tells You

Assuming you made your window wide enough, `fs_usage` provides the following data for each system call:

- A timestamp of when the system call occurred.
- The name of the system call. Table 9.1 lists the most common system calls.
- Additional information that depends on the nature of the system call. Make sure your shell window is wide or the additional information will not appear. Look at Table 9.2 for the types of additional information `fs_usage` provides.
- The pathname of the file the system call accessed.
- The amount of time, in seconds, the program spent in the system call. If the listing has a W next to it, the amount of time includes the time spent waiting for a file operation to finish.
- The name of the process that made the system call. The process name is useful only if you're viewing multiple processes with `fs_usage`. If the application is a CFM application (an executable that runs on Mac OS 9 and X), the name will be LaunchCFMApp. The Xcode Tools do not create CFM applications so you shouldn't have to worry about the name of your application not appearing.

If you use the debug runtime library for your program, `fs_usage` lists the high-level File Manager function calls Cocoa and Carbon programs use to access files. For the File Manager function calls, `fs_usage` displays the parameters passed to each function instead of the extra information in Table 9.2.

**Table 9.1 Common System Calls**

System Call	Description
CACHE_HIT	The computer found the data it needs in the computer's cache. Cache hits are good because the computer doesn't have to go to RAM or disk to retrieve the data. The latest version of <code>fs_usage</code> requires you to use the <code>-f</code> option to report cache hits.
PAGE_IN	The computer moved data from disk to RAM.
PAGE_OUT	The computer moved data from RAM to disk.
read	Reads data from a file.
write	Writes data to a file.
open	Opens a file.
close	Closes a file.
lseek	Moves to a particular point in a file.
fstat	Retrieves information about an open file, such as its size, the last time it was accessed, and the last time its contents changed.
stat	Retrieves the same information as <code>fstat</code> , but the file does not have to be open.
lstat	Retrieves the same information about a file as <code>fstat</code> , but the file does not have to be open. If the file refers to a symbolic link, <code>lstat</code> returns information about the link, not the file to which the link refers.
getattrlist	Returns a list of attributes for a file system object such as a volume, directory, file, or file fork. Some information <code>getattrlist</code> returns for a file includes its name, its size, the number of forks (Mac files have two forks: data fork and resource fork), and the size of each fork.
getdirentries	Returns information about the files and subdirectories inside a given directory.
getdirentriesattr	Returns a list of attributes for multiple directories. It combines the work of <code>getattrlist</code> and <code>getdirentries</code> .
mmap	Maps a file into memory.

**Table 9.2 Extra Information `fs_usage` Provides**

Information	Prefix	Description
Address	A=	For cache hits, page ins, and page outs, the address tells you where the cache hit, page in, or page out occurred in memory.
Bytes	B=	For file reads and writes, the number of bytes the computer read or wrote from the file. For page ins and page outs, the number of bytes the computer paged in from disk or paged out to disk. <code>fs_usage</code> reports the number of bytes as a hexadecimal number.
Disk Block Number	D=	For file reads and writes, the physical disk block number being read from or written to.
File Descriptor	F=	An integer that identifies an open file. <code>fs_usage</code> returns a file descriptor for most of the file operations in Table 9.1.
Offset	O=	For <code>lseek</code> operations, the offset tells you how far the computer moved from the start of the file.
Select Return	S=	The return value of the <code>select</code> system call. A value of zero means select timed out.
Error Number	[ ]	If a file operation produced an error, the error number appears in brackets.

## fs\_usage Options

The `fs_usage` tool provides three options. You can use multiple options in one command. The following command:

```
fs_usage -w -f network AppTitle
```

Combines the `-w` and `-f` options to limit `fs_usage`'s output to network system calls and tells `fs_usage` to wrap its output to a second line if the window isn't wide enough to display all the output on one line.

### -e Option

The `-e` option allows you to exclude certain processes from the `fs_usage` report. Supply a list of process ID numbers or application names.

```
fs_usage -e 449 465 508
```

On Mac OS X the `-e` option does not help much because there are too many processes running. As a test, I started up my Mac, launched the Terminal application, and ran `top`. `top` reported 39 running processes when I had only two applications running: the Finder and Terminal. To look at a few applications, supply the process ID numbers of the applications you're interested in viewing instead of using the `-e` option with every process you don't want to see.

### -f Option

Normally `fs_usage` displays every system call it finds. The `-f` option filters `fs_usage`'s output. Supply one of three filtering modes: network, file system, and cache hit. With the network filtering mode.

```
fs_usage -f network AppTitle
```

`fs_usage` displays only network system calls. With the file system filtering mode.

```
fs_usage -f filesys AppTitle
```

`fs_usage` displays only file system calls.

The latest version of `fs_usage` requires you to use the `-f` option and cache hit mode to report cache hits.

```
fs_usage -f cachehit AppTitle
```

### -w Option

`fs_usage` usually supplies as many columns of data as will fit in your shell window; the wider the window, the more columns of data `fs_usage` shows. When you use the `-w` option, `fs_usage` displays all the available information, wrapping the output to the next line if it won't fit on one line. The `-w` option saves you from having to resize the window to see all of `fs_usage`'s output.

```
fs_usage -w AppTitle
```

## sc\_usage

The `sc_usage` tool maintains a running list of system calls and page faults for a given application. You must use the `sudo` command to run `sc_usage`. Supply either a process ID or the name of a currently running application to `sc_usage`.

```
sc_usage ProcessID
```

```
sc_usage AppTitle
```

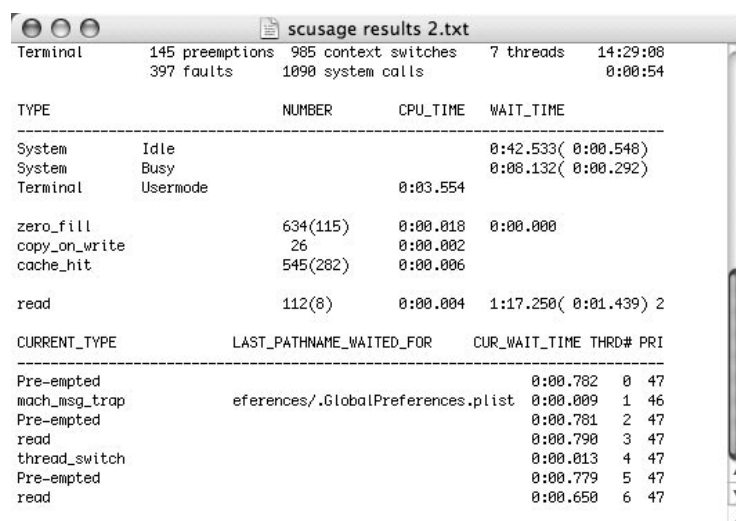
When you run `sc_usage` on your application, the chances are high no data will appear initially because you're in the Terminal application instead of your application. Switch to your application and do some things in it to see some of the data `sc_usage` records.

## What sc\_usage Tells You

`sc_usage` provides its output differently than `fs_usage`. `fs_usage` has one listing for each individual event while `sc_usage` has one listing for each type of system call along with the number of calls. If your application has 25 cache hits, `fs_usage` displays 25 `CACHE_HIT` listings while `sc_usage` displays one `cache_hit` listing with a count of 25.

By default `sc_usage` updates its data every second, replacing the old information with the new. This behavior differs from `fs_usage`, which adds each system call to the output immediately. The `sc_usage` output is more useful to view while your program's running rather than saving and viewing later.

`sc_usage` provides its output in three areas as you can see in Figure 9.1. At the top is summary information about the program being monitored. In the center is a list of the system calls the program made along with information about the calls. At the bottom is a list of blocked system calls along with information about each call.



Summary Statistics				
Terminal	145 preemptions	985 context switches	7 threads	14:29:08
	397 faults	1090 system calls		0:00:54

TYPE	NUMBER	CPU_TIME	WAIT_TIME
System Idle			0:42.533( 0:00.548)
System Busy			0:00.132( 0:00.292)
Terminal Usermode		0:03.554	
zero_fill	634(115)	0:00.018	0:00.000
copy_on_write	26	0:00.002	
cache_hit	545(282)	0:00.006	
read	112(8)	0:00.004	1:17.250( 0:01.439) 2

CURRENT_TYPE	LAST_PATHNAME_WAITED_FOR	CUR_WAIT_TIME	THR#	PRI
Pre-empted		0:00.782	0	47
mach_msg_trap	ferences/.GlobalPreferences.plist	0:00.009	1	46
Pre-empted		0:00.781	2	47
read		0:00.790	3	47
thread_switch		0:00.013	4	47
Pre-empted		0:00.779	5	47
read		0:00.650	6	47

**Figure 9.1**

`sc_usage`'s output.

## Program Summary Information

At the start of the report is summary information about the program being monitored. `sc_usage` reports eight pieces of summary information.

- The program's name.
- The number of preemptions. A preemption occurs when the operating system interrupts your program to give time to another program. Preemptions are good as a Mac user. They allow you to simultaneously write source code in Xcode, download a file in Safari, and listen to a CD in iTunes.
- The number of page faults. A page fault occurs when the operating system can't find a page of memory in physical memory.
- The number of context switches. A context switch occurs when the operating system switches to another program. A major context switch occurs when you switch applications. If you're in Safari and switch to iTunes, a major context switch changes the foreground process from Safari to iTunes. A minor context switch occurs when the operating system gives time to a program running in the background. If you're surfing the Internet on Safari and listening to music with iTunes, a minor context switch gives iTunes the time it needs to play the music you're listening to.
- The number of system calls your program made.
- The number of threads in the program.
- The current time. It's always important to know what time it is.
- The elapsed time, the amount of time `sc_usage` has been monitoring your program.

When you look at the summary information, remember that the numbers of preemptions, page faults, context switches, and system calls `sc_usage` reports are the numbers that occurred during the last sampling period. If you run your program and switch to `sc_usage`, you could see your preemptions, page faults, context switches, and system calls trickle down to zero. The trickling occurs because `sc_usage`'s default sampling period is one second. If you're examining `sc_usage`'s results, your application won't have much activity in the previous second.

## System Call List

After the program summary information is a list of each system call your application makes. `sc_usage` tells you the following information for each system call:

- The system call name.
- The number of times your program made the system call.
- The amount of time your program spent in the system call.
- The amount of time the call has been waiting.

When reporting the number of times your program made a system call and the amount of time the call has been waiting, there may be two values, one of which is in parentheses.

```
500 (41)
```

The listing above says your program made this system call 500 times since launching `sc_usage`. 41 of the 500 calls occurred during the last sampling period. The first three listings in the system call list are.

- System Idle, which tells you the amount of time the operating system is idle.
- System Busy, which tells you the amount of time the operating system is busy.
- Usermode, which tells you the amount of time spent in your program.

Table 9.3 Page Fault Types

Page Fault Type	Description
cache_hit	The operating system found the page of memory in the computer's cache.
page_in	The operating system moved the page from disk to physical memory.
zero_fill	The operating system created the page of memory and filled it with zeroes.
copy_on_write	The operating system copied the memory page from another page of memory.

After the System Idle, System Busy, and Usermode listings comes the page fault system calls. Table 9.3 lists these calls.

After the page fault system call listings come the rest of the system call listings. You can see some common system calls in Table 9.1.

### Blocked System Call List

After the system call list comes a list of blocked system calls, calls that are waiting for an event to occur. For each blocked system call, `sc_usage` reports.

- The system call name.
- The path name of the file referenced by the system call. Not every system call references a file.
- The amount of time the system call has been waiting.
- The thread number.
- The current scheduling priority. Higher numbers mean higher priority.

The most common blocked system call is `mach_msg_trap`. When your program is waiting for the user to do something, it calls `mach_msg_trap` many times.

### sc\_usage Options

`sc_usage` has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
sc_usage -l -s 10 AppTitle
```

Combines the `-l` and `-s` options. It tells `sc_usage` to turn off continuous updating of data and to use a sampling interval of ten seconds.

### -c Option

The `-c` option lets you specify a code file that contains the system calls you want `sc_usage` to report your program making. You can view the default code file at `/usr/share/misc/trace.codes`. Supply the path to the code file when using the `-c` option.

```
sc_usage -c CodeFile AppTitle
```



## -e Option

The `—e` option sorts the output data by the call count, the number of times your application made the system calls. By default `sc_usage` sorts the listings by the amount of time your program spent in each system call.

```
sc_usage -e AppTitle
```

`sc_usage` sorts the system calls on a first come, first served basis. When your program makes a system call for the first time, the call appears at the bottom of the list. If your program makes multiple system calls for the first time in a given sampling period, `sc_usage` sorts the calls based on the number of times your program made them, and this order never changes. Suppose you have two system calls, A and B. If during the first sampling period, your program calls A 10 times and B 20 times, B appears ahead of A. If your program calls A 100 times the next period and doesn't call B, B still appears ahead of A even though A has been called more than B.

## -E Option

Running `sc_usage` with the `—E` option launches your program first. Use the `—E` option when you want to see the system calls your program makes when it starts. Supply a path to your program's executable file and any runtime arguments your program needs to run. For Mac OS X application bundles, the executable file resides three directories inside the application bundle.

```
Application Bundle
  Contents
    MacOS
      Executable File
```

If you're in the directory where the application bundle resides, you would launch the program with the following command:

```
sc_usage -E AppTitle.app/Contents/MacOS/AppTitle
```

## -l Option

The `—l` option turns off the continuous updating of data. Every time `sc_usage` updates its data, it appends the output to the shell window. If you use the default sampling rate of one second and watch your program for one minute, you would end up with 60 reports in the shell window. Use the `—l` option if you want a record of all system calls your application makes.

## -s Option

The `—s` option lets you specify how often you want `sc_usage` to update its output. Supply the number of seconds you want in the sampling period; the default period is one second. The following command tells `sc_usage` to update its output every five seconds:

```
sc_usage -s 5 AppTitle
```

## vmmap

The `vmmap` tool gives you a map of the virtual memory the operating system reserved for your program. Supply a process ID number when running `vmmap`.

```
vmmap ProcessID
```

## What vmmap Tells You

The report `vmmap` generates has three sections.

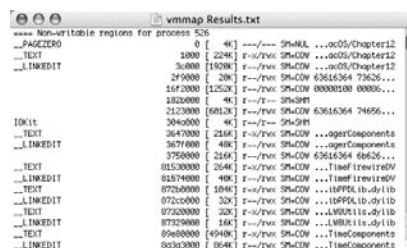
- The non-writable memory regions, which consist mostly of shared libraries to which you linked your application. Shared libraries being non-writable is a good thing. If you write an application that uses QuickTime, you don't want your application to be able to overwrite the memory QuickTime is using.
- The writable memory regions.
- A summary report of the virtual memory map.

When you look at the report, keep in mind that it's telling you about the memory the computer reserved for your application. It does not necessarily mean your application is using all of that memory. When the operating system launches your program, it reserves a large chunk of memory, more than most programs need. From this large chunk of reserved memory, the operating system allocates smaller chunks when your application needs them. The `vmmap` report gives you a map of the initial large chunk of reserved memory. You have to do some digging to discover how much of the reserved memory your program uses.

## Non-Writable Memory Regions

The report for the non-writable memory regions your application reserves has one listing for each memory region. Figure 9.2 shows a portion of `vmmap`'s report for non-writable memory regions; a complete listing is too large to fit in a shell window. For each memory region, `vmmap` tells you the following information:

- The purpose of the memory in the region.
- The starting address of the memory region. If you're debugging your program, you can look at this address in the debugger to view the memory contents.
- The size of the region. It appears in brackets.
- The permissions for the memory region.
- The sharing mode of the memory region.
- Additional data that depends on the memory region. Some regions have no additional data to display. For some regions `vmmap` shows the pathname of the executable file for the memory region. Most of these pathnames will be libraries linked to your program. For some regions `vmmap` shows the contents of the memory in the region. The amount of data that appears in the window depends on the width of your window. Wider windows reveal more memory contents and a larger portion of pathnames.



```

vmmap Results.txt
==== Non-writable regions for process 526
...PAGEZERO
...TEXT
...LINKEDIT
0 [ 4K] r--/--- SHARED .../System/Library/Frameworks/Chapter12
1000 [ 224K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
3000 [ 180K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
249000 [ 20K] r-/rwx SHARED 63616364 73626...
1642000 [ 1252K] r-/rwx SHARED 00000100 00006...
1820000 [ 4K] r--/r-- SHARED
2123000 [ 6012K] r-/rwx SHARED 63616364 74656...
3040000 [ 4K] r-/r-- SHARED
3647000 [ 216K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
3670000 [ 40K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
3750000 [ 216K] r-/rwx SHARED 63616364 68626...
8153000 [ 304K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8157000 [ 40K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8720000 [ 104K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8721000 [ 32K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8722000 [ 32K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8723000 [ 16K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
8960000 [ 404K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12
9030000 [ 864K] r-/rwx SHARED .../System/Library/Frameworks/Chapter12

```

**Figure 9.2**

`vmmap` output for a non-writable memory region.

## Region Purpose

The region purpose of a memory region gives you a general idea of the region's contents. The first non-writable memory region for every application has the name `__PAGEZERO` as its purpose. This region is protected; accessing it will crash your program.

Non-writable memory regions have six common purposes.

- `__TEXT` regions contain executable code or constant data like the constants your application declares.
- `__DATA` regions contain data, which comes as a shock to you. The data is read-only for a non-writable memory region, but a writable memory region would contain data that your application could both read and write.
- `__LINKEDIT` regions contain raw data the dynamic linker uses, such as symbol table entries. These regions generally follow `__TEXT` regions.
- `__OBJC` regions contain data that the Objective C runtime library uses. Only Objective C programs have these regions.
- Shared memory regions are shared by multiple applications. System libraries like Cocoa and Carbon are the major source of shared memory regions.
- Mapped file regions are memory mapped files, where the operating system maps part of a file into a program's virtual address space. Memory mapped files let multiple programs read from and write to the same file.

One not so common purpose deserves some explanation. Guarded memory regions, which show up as `GUARD` in a `vmmmap` report, prevent out of order accesses. Normally, the computer allows operations to be performed out of order. Suppose your program is currently performing a series of integer calculations, followed by some floating-point calculations. If the CPU performed the operations in order, the floating-point unit would be idle until the series of integer calculations finished. To make use of the idle floating-point unit, the CPU performs the floating-point calculations while it performs the series of integer calculations. The floating-point calculations are out of order operations in this situation. Out of order operations are normally good; they allow your program to run efficiently. Sometimes accessing memory out of order can wreak havoc with your program. Memory that controls I/O devices is especially vulnerable to out of order memory accesses. Making vulnerable memory regions guarded prevents bad things from happening in your program.

## Permissions

The permissions for a memory region tell you how you can access the region. `vmmmap` lists two sets of permissions for each memory region. The first set lists the permissions for your application, and the second set lists the maximum permissions. The most common use of maximum permissions is running your program in a debugger. In a non-writable memory region the most common permission is.

```
r--/rwx
```

Which means your application can read memory in this region, but cannot write to memory or execute memory in this region. Your application will never have write permission in non-writable memory regions. The only way a memory region can have execute permission is if the operating system loaded a piece of executable code in the region. An application with maximum permission can read, write, and execute memory in this memory region. The most common situation to use maximum permissions to write to memory in a non-writable region is a debugger inserting a breakpoint in an application.

If you look at a `vmmmap` report, you can see the first listing, `__PAGEZERO`, has six dashes in the permissions area, which means nobody can read, write, or execute memory there. If you've done any work with pointers, you know that accessing null pointers will crash your program. Denying permission to access `__PAGEZERO` ensures a crash if you do anything with a null pointer.

## Sharing Modes

The sharing mode of a memory region tells you how the region interacts with other programs. A memory region can have one of seven possible sharing modes.

- Copy on write (COW) regions can be shared at multiple locations in your application or shared by multiple applications. When your application modifies the memory of a copy on write region, it receives a copy of the region, and the page of memory your application modified becomes private. When all of the pages of a memory region become private, the region becomes private as well.
- Private (PRV) regions are visible only to your application.
- Empty (NUL) regions do not exist in physical memory.
- Aliased (ALI) regions point to another memory region.
- Shared (SHM) regions are shared by multiple applications.
- Zero filled (ZER) regions have had their contents cleared and filled with zeroes.
- Shared alias (S/A) regions combine the characteristics of aliased and shared memory regions.

## Writable Memory Regions

`vmmap` provides the same types of information for writable memory regions as it does for non-writable ones: purpose, starting address, size, permission, sharing mode, and possible additional data. When you look at the permissions for a writable memory region, you will see most of the listings give your application write access, which is what you expect for a writable memory region. If a writable memory region shows no additional data, the memory in the region is heap memory or stack memory.

Writable memory regions rarely have the `__TEXT` and `__LINKEDIT` purposes that are common in non-writable regions. Common purposes for writable memory regions include.

- `__DATA` regions contain data (surprise) that your application can both read from and write to.
- `MALLOC` regions have been allocated using the function `malloc()`.
- `MALLOC_TINY` regions consist of small memory allocations, allocations 512 bytes and smaller.
- `MALLOC_LARGE` regions consist of large memory allocations. Apple's documentation says large memory allocations are at least 4 pages in size, 16KB. But I saw `MALLOC_LARGE` regions that were 4KB when I ran `vmmap`.
- `MALLOC_FREE` regions were allocated earlier, used, and freed when your program was finished with the memory.
- `VM_ALLOCATE` regions are regions that were allocated with the `vm_allocate()` function. This function allocates virtual memory regions from which `malloc()` allocates memory.
- `STACK` regions contain stack memory, which is where the computer places the parameters of every function your application calls.
- `ATS` regions involve the Apple Type Services (ATS) framework, which deals with fonts. If your program works with text, chances are high you'll have `ATS` memory regions.
- `__OBJC` regions contain data that the Objective C runtime library uses.
- `__LOCK` regions are locked, which means the operating system can't move them when memory runs low.

You might be wondering where your program calls `malloc()` if you didn't call it in your code. When you use the `new` operator in a C++ program to create an object, you're indirectly calling `malloc()`. When you use the Carbon and Cocoa functions to create things like windows, menus, dialog boxes, sounds, and graphics, you're calling `malloc()` indirectly.

## Summary Report

For non-writable memory regions, `vmmmap`'s summary displays the following data:

- The total amount of memory reserved for your application.
- The amount of resident memory, the memory that is in physical RAM.
- The amount of memory that was either swapped out or unallocated. Swapped-out memory is memory that was in RAM, but moved to disk because other programs needed the RAM.

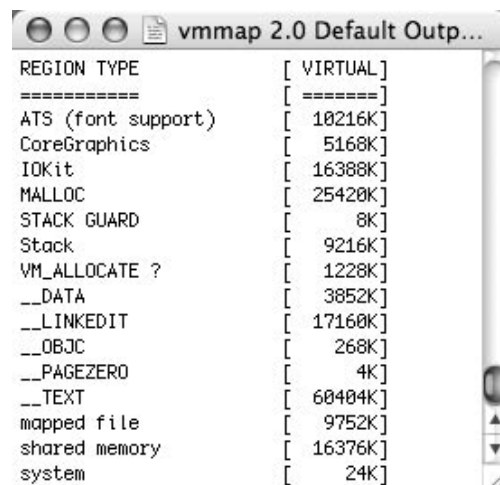
Remember that shared libraries your application links to comprise most of the read-only memory regions. Other applications use these libraries as well so seeing a high amount of resident memory does not necessarily mean your application is a memory hog.

For writable memory regions, `vmmmap`'s summary tells you the following information:

- The total amount of memory reserved for your application.
- The amount of memory the computer wrote in your application.
- The amount of resident memory.
- The amount of swapped-out memory. Swapped-out memory can mean one of two things: you're switching to other applications or your program uses a lot of memory.
- The amount of unallocated memory.

The sum of resident memory and swapped-out memory is the best measurement of the amount of memory your program uses.

At the end of the summary report, `vmmmap` breaks down the memory map by region purpose and tells you the total amount of virtual memory for each purpose. Figure 9.3 shows a sample breakdown.



REGION TYPE	[ VIRTUAL ]
=====	[ ===== ]
ATS (font support)	[ 10216K ]
CoreGraphics	[ 5168K ]
IOKit	[ 16388K ]
MALLOC	[ 25420K ]
STACK GUARD	[ 8K ]
Stack	[ 9216K ]
VM_ALLOCATE ?	[ 1228K ]
__DATA	[ 3852K ]
__LINKEDIT	[ 17160K ]
__OBJC	[ 268K ]
__PAGEZERO	[ 4K ]
__TEXT	[ 60404K ]
mapped file	[ 9752K ]
shared memory	[ 16376K ]
system	[ 24K ]

**Figure 9.3**

Memory region breakdown.

## vmmap Options

vmmap has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
vmmap -w -submap ProcessID
```

Tells vmmap to print wide output and to print submap information in the output.

### -d Option

When you use the `-d` option, vmmap takes two snapshots of your program's virtual memory map. It takes the first snapshot immediately and takes the second one after a period of time that you specify. After taking the snapshots, vmmap reports three pieces of information.

- The memory regions that changed in the second snapshot.
- Memory regions in the first snapshot that aren't in the second.
- Memory regions in the second snapshot that aren't in the first.

The vmmap report has one difference listing for each memory region that changed between the first snapshot and the second. The difference listing starts with the text `Diff`, then displays the region as both snapshots recorded it. The usual cause of a difference in a memory region is a change in sharing mode.

After showing the memory regions that changed, vmmap lists the memory regions that appear in only one of the snapshots. It starts with the regions that appear only in the first snapshot then lists the regions that appear only in the second snapshot. Non-writable memory regions appear before writable ones.

Using the `-d` option is relatively simple. Supply the number of seconds you want vmmap to wait before taking the second snapshot. The following example waits 30 seconds for a program with process ID 555:

```
vmmap -d 30 555
```

### -w Option

The `-w` option displays wide output for each memory region. Wide output allows you to see a greater amount of additional data for each region.

```
vmmap -w ProcessID
```

### -resident Option

When you run vmmap with the `-resident` option, it lists the virtual and resident size of each memory region. The resident size tells you how much of the region is in physical memory.

```
vmmap -resident ProcessID
```

### –pages Option

When you run `vmmap` with the `–pages` option, it lists the size of each memory region in pages instead of bytes. Memory pages on PowerPC Macs are 4KB. The default page size is also 4KB on Intel Macs, but pages can also be 2MB and 4 MB.

```
vmmap –pages ProcessID
```

### –interleaved Option

When you run `vmmap` with the `–interleaved` option, the report does not separate the memory regions into writable and non-writable regions. It lists the memory regions in order of their starting address, with the lowest memory regions appearing first.

```
vmmap –interleaved ProcessID
```

### –submap Option

The `–submap` option tells `vmmap` to print information about memory submaps. A *submap* is an area of memory the operating system can use in multiple applications. A Cocoa runtime library may reside in a submap so any running Cocoa application can use it.

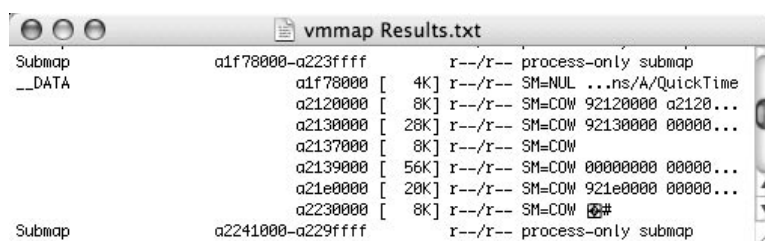
In a `vmmap` report all listings below a submap listing belong to that particular submap until another submap listing appears. Figure 9.4 shows an example of a submap listing.

```
vmmap –submap ProcessID
```

### –allSplitLibs Option

The `–allSplitLibs` option tells `vmmap` to print information about all shared system split libraries. The default behavior is to print information about only the libraries your program loads. *Shared system split libraries* are dynamic libraries that multiple programs share. Examples of these libraries are the Cocoa and Carbon libraries. The `–allSplitLibs` option generates a lot of unused split lib listings for non-writable regions.

```
vmmap –allSplitLibs ProcessID
```



**Figure 9.4**

Listing for a submap.

## heap

The `heap` tool lists the objects allocated in your application's heap. `MallocDebug`, which I covered in Chapter 7, provides similar information. `heap` works better with command-line programs and its output is easier to look at later. Supply a process ID when running `heap`.

```
heap ProcessID
```

The output `heap` generates comes in three levels. First, `heap` reports a summary for each memory zone. The summary tells you the following information for each zone:

- The size of the zone.
- The number of nodes allocated.
- The amount of memory allocated.
- The largest unused block of memory in the zone.

After reporting each memory zone, `heap`'s summary reports the total number of nodes allocated with the total amount of allocated memory. What does it mean when a node is allocated? It means your program made a memory allocation. One node equals one memory allocation.

Second, `heap` reports the sizes of each memory allocation. For each memory zone `heap` tells you the number of nodes allocated along with the size listings. In each size listing, `heap` reports the size of the memory allocation with the number of allocations in brackets. The following listing:

```
24KB [ 7 ]
```

Says your program made seven memory allocations of 24KB in the memory zone.

`heap` sorts the size listings by allocation size, with the largest allocations coming first. After listing the information for each zone, `heap` reports the same information for all zones combined.

Finally, `heap` reports the Objective C classes your program allocated. The report starts with a total number of Objective C classes allocated on the heap for the whole application. Then for each memory zone, it lists the classes the computer allocated for that zone. For each class `heap` tells you.

- The class name.
- The number of objects of the class your program allocated.
- The amount of memory allocated.

`heap` sorts the listings by the number of allocations, with the highest numbers appearing first in the list.

The latest version of `heap` has a new option for Objective C programs. The `--guessNonObjects` option tells `heap` to search the memory of each object for pointers to non-objects, blocks of memory allocated by `malloc()`. The `heap` report has one listing for each of these memory blocks. The block listings list the Objective C object, with the offset from the start of the object in brackets. The following listing:

```
NSCFArray[ 36 ]
```

References a block of memory that is located 36 entries from the start of the array.



## leaks

The `leaks` program checks your application for memory leaks, something `MallocDebug` also does. Even though `MallocDebug` is superior in most cases, you should check your program in `leaks` as well. `MallocDebug` does not detect memory leaks in circularly linked (A is linked to B, B is linked to C, and C is linked to A) data structures and does not detect memory leaks in groups of connected memory nodes. `leaks` detects any memory leaks in both of these situations. Run your program through `leaks` after `MallocDebug` says the program is free of memory leaks.

`leaks` has one major flaw. It does not detect memory leaks for memory allocated by the Carbon function `NewHandle()`. Use `MallocDebug` to detect `NewHandle()` memory leaks.

## Running leaks

You can run `leaks` without switching to `root`. All you must supply is a process ID or application name.

```
leaks AppTitle
leaks ProcessID
```

Setting the environment variable `MallocStackLogging` to 1 turns on call stack recording, which tells `leaks` to display the call stack for each memory leak. Turning on call stack recording is a good idea. If you have a memory leak in your program, the first thing you want to know is where you're leaking memory so you can fix the leak in your code. By turning on call stack recording, `leaks` can tell you where you're leaking memory.

## What leaks Tells You

`leaks` provides two pieces of information about your program. The first piece of information is a summary that reports the following information:

- The number of nodes your application allocated, which is the number of memory allocations your program made.
- The amount of memory your program allocated.
- The number of memory leaks.
- The total amount of leaked memory.

The second piece of information is a listing for each memory leak that `leaks` found. Each listing reports the following information:

- The address of the leaked memory.
- The size of the leak.
- A memory dump of the leak. For memory leaks 128 bytes and smaller, `leaks` shows the contents of the leaked memory. For memory leaks larger than 128 bytes, `leaks` displays the first 128 bytes of leaked memory.
- If the leak occurs in a Core Foundation or Cocoa class, `leaks` reports the name of the class.
- If you set the `MallocStackLogging` environment variable, `leaks` displays the call stack of functions leading to the memory leak.

## leaks Options

`leaks` has three options you can supply. You can use multiple options. The following command:

```
leaks --nocontext --nostacks
```

Tells `leaks` to conceal the memory dump and call stack in the individual memory leak listings.

### -nocontext Option

The `--nocontext` option tells `leaks` to suppress the memory dump. Use the `--nocontext` option if you don't care about the contents of the memory your program is leaking.

```
leaks --nocontext AppTitle
```

### -nostacks Option

Setting the environment variable `MallocStackLogging` to 1 tells `leaks` to display the call stack for each leak. The `--nostacks` option tells `leaks` to suppress the call stack display. I'm not sure why you would want to suppress the call stack display; the call stack display helps you discover where the memory leaks are in your code.

```
leaks --nostacks ProcessID
```

### -exclude Option

When you use the `--exclude` option, `leaks` reports the summary information: the amount of memory allocated, the number of memory leaks, and the total amount of leaked memory. If the function you supply appears in the call stack of a memory leak, `leaks` does not create a listing for that memory leak. Use the `--exclude` option to keep functions you already know leak memory and functions falsely accused of leaking memory from creating unnecessary memory leak listings.

```
leaks --exclude FunctionName ProcessID
```

## malloc\_history

As its name suggests, `malloc_history` supplies a history of every memory allocation your application makes using the `malloc` library. `MallocDebug`, covered in Chapter 7, provides similar information, but `malloc_history` works better with command-line programs and makes saving the results to a text file easier.

To run `malloc_history` on your application, you must turn on the `malloc` library's debugging capabilities by setting the environment variable `MallocStackLogging` to 1. While you're setting environment variables, you may also want to set the variables `MallocStackLoggingNoCompact` and `MallocScribbling` to 1. Setting the `MallocStackLoggingNoCompact` environment variable allows you to run `malloc_history` on a specific memory address. Setting the `MallocScribbling` environment variable causes the operating system to overwrite memory your application frees. Overwriting freed memory helps you detect memory smashers, places in your code that overwrite memory.

### Running malloc\_history

There are two ways to run `malloc_history` on an application.

```
malloc_history ProcessID -all_by_size  
  
malloc_history ProcessID -all_by_count
```

`malloc_history` displays the memory allocations your application made in the shell window with one listing for each combination of allocation size and call stack. A listing tells you the following information:

- The thread.
- The number of memory allocations.
- The size of each allocation.
- The call stack for the allocation.

The only difference between `-all_by_size` and `-all_by_count` is how `malloc_history` sorts the listings. `-all_by_size` sorts by allocation size, with the largest sizes appearing first while `-all_by_count` sorts by allocation count, with the largest counts appearing first.

### Running malloc\_history on a Specific Memory Area

`malloc_history` normally lists every memory allocation your program makes. You can tell `malloc_history` to report allocations made in a particular memory area by supplying a memory address to `malloc_history`.

```
malloc_history ProcessID MemoryAddress
```

`malloc_history` provides a history of all memory allocations that manipulated memory at the address you specified. For each memory allocation, `malloc_history` reports the size of the allocation (It says `arg =`), the thread, and the call stack.

To run `malloc_history` on a memory address instead of an entire program, you must set the environment variable `MallocStackLoggingNoCompact` to 1.

# Chapter 10

## gcov

`gcov` is a command-line program that keeps track of how many times your program executes each line of code in the program. Code testing is `gcov`'s main purpose. With `gcov` you can make sure each line of code executes at least once, which goes a long way toward making your code stable and reliable.

### Generating Code Coverage Data in Xcode

Before you can run `gcov` with your source code files, you must tell the compiler to generate code coverage data. To make your source code files ready for `gcov`, you must compile your program with the compiler flags.

```
-fprofile-arcs  
  
-ftest-coverage
```

The `-fprofile-arcs` flag tells the compiler to perform code coverage, tallying the number of times your program executes each line of code. The `-ftest-coverage` flag creates the data files `gcov` uses. To generate code coverage data in Xcode:

- 1) Open the information panel for your project by selecting the project name from the Groups and Files list and clicking the Info button.
- 2) Click the Build tab in the information panel. Some versions of Xcode have a Styles tab.
- 3) Choose Code Generation from the Collection pop-up menu.
- 4) Select the Instrument Program Flow checkbox.
- 5) Select the Generate Test Coverage Files checkbox.
- 6) Choose None from the Optimization Level setting's pop-up menu.

Some versions of Xcode do not have the Instrument Program Flow and Generate Test Coverage Files checkboxes. If your version of Xcode doesn't have these checkboxes, you must manually set the `-fprofile-arcs` and `-ftest-coverage` flags. Use either the Other C Flags setting or the Other C++ Flags setting to set the flags. These settings are in the Language collection. The setting you use to set the flags depends on the language you're using. Objective C programs should use the Other C Flags setting. Objective C++ programs should use the Other C++ Flags setting. To manually set the flags:

- 1) Select either the Other C Flags setting or the Other C++ Flags setting.
- 2) Click the Edit button.
- 3) A sheet opens. Add the `-fprofile-arcs` and `-ftest-coverage` flags.
- 4) Click the OK button.

If you're compiling your program with `gcc 4`, you must build your program with the linker flag `-lgcov`. In Xcode add the `-lgcov` flag to the Other Linker Flags build setting, which is in the Linking collection.

You should turn off optimization when running your application with `gcov` so `gcov` can provide an accurate report of which lines of code in your program execute. Compiler optimizations may change the flow of your program and eliminate variables. In addition they may add, move, and eliminate statements from your code. These optimizations make it hard to determine which lines of your code are executing when the program runs.

You should have only one statement per line of source code when using `gcov`. If you have a line of code with multiple statements, such as the following line:

```
if (x == 0) y = 3;
```

`gcov` reports the number of times the `if` statement executed, but you have no way of knowing how many times the statement `y = 3` executed. Complicated macros that contain multiple statements, loops, or conditional statements have the same problem. `gcov` reports only the macro call, not individual statements inside the macro.

## Running `gcov`

After telling Xcode to generate code coverage data, recompile your program and run it to measure code coverage.

- 1) Clean your project by choosing Build > Clean.
- 2) Build your project by choosing Build > Build.
- 3) Run your application.

For each source code file in your project, the compiler creates two files with the extensions `.gcno` and `.gcda`. If you build your program with `gcc 3`, the compiler creates three files with the extensions `.bb`, `.bbg`, and `.da`. If you have a file in your program called `main.c`, the compiler would create the files `main.gcno` and `main.gcda`. `gcov` uses these files to generate the reports you use to see which lines of code are executing. Xcode places the `.gcno` and `.gcda` files in the same folder as your project's object files. To find these files, go to your project's folder. The files will be in the following directory:

```
build/ProjectName.build/Debug/ProjectName.build/Objects-normal/ppc
```

On Intel Macs the last directory will be `i386` instead of `ppc`. Release builds will have a `Release` directory instead of a `Debug` directory.

After running your application, you're ready to run `gcov`. Because `gcov` is a command-line tool, you must launch the Terminal application first. Move to the directory containing the `.gcno` and `.gcda` files. Refer to the section "A Command Line Primer" in Chapter 9 for more information on navigating directories from the command line. Each source code file in your program has corresponding files with the extensions `.gcno`, and `.gcda`. To run `gcov`, supply the name of the source code file with no extensions. The example below demonstrates how you would run `gcov` on a file named `main.c`.

```
gcov main
```

When you run `gcov` it tells you the percentage of source code lines that executed when you ran your application. It also creates a file with the extension `.gcov` (`main.c.gcov` in the example above). This file contains the data `gcov` collected on the source code file. Run `gcov` on the rest of the files in your program. You can run `gcov` on only one file at a time.

## Interpreting gcov's Results

To read the reports that gcov generated, open any of the .gcov files in a text editor. When you look at the .gcov file for one of your source code files, you'll notice the .gcov file contains the code from your source code file with numbers running along the left side. The number next to a line of code tells you how many times your program executed the line of code. If a line of code did not execute, ##### appears instead of a number. From a testing standpoint what's most important is whether or not your program executed a line of code, not the number of times the line executed.

gcov's numbers can help measure your code's performance, especially if you use gcov with gprof. gprof reports the functions where your program spends the most time. You can examine those functions in gcov to find the lines of code that execute the most. How do some lines of code in a function execute more than others? Loops (for and while) and conditional statements (if-then-else and switch) cause some lines of code to execute more than others. Code inside loops executes more than code outside them. Code inside a conditional statement executes less than other lines of code. Suppose you have a switch statement with five cases. When your program reaches the switch statement, it can execute the code for only one of the five cases, which means the code for the five cases executes less than the switch statement.

## Running Multiple Tests

Unless you have a very small program, you won't be able to test every line of code by running the program once. For complete testing you should run a series of tests, with each test covering a piece of your code. With gcov you can run the series of tests, then check the .gcov files to make sure you tested all of the code.

Conducting multiple tests on your code requires no effort on your part. As long as you don't delete the .gda files, gcov appends the results to the previously calculated results. If a line of code executes 1000 times the first time you run your program and executes 600 times the next time you run it, gcov reports the number 1600. All you have to do to conduct multiple tests is run your program multiple times. Check the .gcov files for the dreaded #####. Keep running your program until it executes all the code.

## gcov Options

Like most Unix command-line tools, gcov has many customization options. You can combine multiple options in one gcov call. The following call:

```
gcov -b -p Filename
```

Combines the -b and -p options to provide branch probability information and preserve path information.

### -a Option

You must compile your program with gcc 4 to be able to use the -a option. The -a option tells gcov to write execution counts for each basic block in a line of code. A basic block is a sequence of instructions that executes in order with no chance of branching. One statement in C, C++, or Objective C consists of multiple assembly language instructions so one statement can have multiple basic blocks. The default gcov behavior is to report the execution count for only the main block in a line of code.

## -b Option

The `-b` option provides branch probability information. When you run `gcov` with the `-b` option, it tells you the following summary information for the source code file:

- The percentage of source code lines executed.
- The percentage of branches executed.
- The percentage of branches taken at least once.
- The percentage of function calls executed.

The output file `gcov` generates contains additional information as well. The extra information involves branches and function calls in your code. The easiest way to explain is with an example.

```

                                case kMoveLeft:
385                                CalculateXSpeed();
call 0 returns = 100%
385                                xVelocity = GetXSpeed();
call 0 returns = 100%

385                                if (CanMoveLeft(xVelocity))
call 0 returns = 100%
branch 1 taken = 14%
332                                MoveLeft(xVelocity);
call 0 returns = 100%
branch 1 taken = 100%
                                else
53                                MoveLeft(HowFarLeft());
call 0 returns = 100%
call 1 returns = 100%

385                                break;
branch 0 taken = 100%
```

The lines with the text `call x returns` and `branch x taken` are what the `-b` option creates. Looking at the listing, you can see every function call returns 100 percent of the time. Most functions your program calls return 100 percent of the time.

What's most interesting is the `branch 1 taken 14%` line. The line says the program took the `else` branch — the condition in the `if` statement was false — 14% of the time. You could also determine the program took the `else` branch 14% of the time by taking 53 (the number of times the `MoveLeft()` call in the `else` block executed) and dividing by 385 (the number of times the `if` statement executed). The other two branches — leaving the `if-then-else` block when the condition in the `if` statement is true and the `break` statement — are taken every time because they are both unconditional branches. With the `-b` option `gcov` generates the branch and function call information for all your code.

If you use `gcc 4` to build your program, a listing precedes each function in `gcov`'s report. The listing tells you the number of times your program called the function, the percentage the function returned, and the percentage of basic blocks that were executed.

## -c Option

The `-c` option differs from the `-b` option in that it records branch information by the number of branches taken instead of the percentage of branches taken. I found no difference between running `gcov` with the `-c` option and running it with no options.

## -f Option

The `-f` option provides a function summary in the shell window when you run `gcov`. The function summary tells you the percentage of source code lines executed for each function in the source code file. The output file `gcov` generates is the same as the default one.

## -l Option

The `-l` option creates long file names for header files that contain source code. The most common case of a header file containing source code involves accessor functions, functions used to access data members of a class. Because accessor functions have only one line of code, some programmers include the code for the function when declaring the function in the class, giving the header file source code.

```
int GetX(void)    { return x; }
```

If you have an Objective C file `MyClass.m` with a header file `MyClass.h`, running `gcov` with the `-l` option.

```
gcov -l MyClass
```

Creates a file named `MyClass.m.MyClass.h.gcov` for the header file. Without the `-l` option, the output file for the header would be `MyClass.h.gcov`. The `-l` option is useful if several files include a header file; it allows you to differentiate which file included the header.

## -n Option

Running `gcov` with the `-n` option tells `gcov` not to generate an output file. General statistics still appear in the shell window. The `-n` option is most useful when combined with the `-b` or `-f` options; otherwise all the `-n` option reports is the percentage of lines executed.

## -o Option

The `-o` option lets you specify a directory where the `.gcno` and `.gcda` files reside. Use the `-o` option if you don't want to manually move to the directory containing the `.gcno` and `.gcda` files. If you wanted to run `gcov` on the file `main.c` and you moved its corresponding `.gcno` and `.gcda` files to the `Code` folder in the `Documents` folder of your hard drive, you would run `gcov` with the following command:

```
gcov -o /Documents/Code main
```



## **-p Option**

The `-p` option preserves the path information in the names of the `.gcov` files `gcov` makes. Preserving path information can help if the source code files for your project reside in multiple directories. Suppose you have a directory called `Code` on your hard drive with networking code files in a directory called `Network`. If you ran `gcov` with the `-p` option on a file in the `Network` folder called `Server.c`, the resulting output file would have the name.

```
Code#Network#Server.c.gcov
```

## **-u Option**

The `-u` option has been added to the latest version of `gcov`. You must compile your program with `gcc 4` to be able to use the `-u` option. It works with the `-b` option. The `-u` option tells `gcov` to include the branch probabilities of unconditional branches.

```
gcov -b -u Filename
```

# Chapter 11

## Version Control with cvs

A version control system tracks the changes made to a file and who made the changes. Version control comes in handy on large projects with multiple developers. Each developer can take an individual file, add code to the file, and the version control program records the code each developer added to the file. Even if you're working on a project by yourself, version control can help you. If you've ever mistakenly saved a file and wished you could go back to the way the file was before you saved it, you'll appreciate version control. With version control you can go back to an older version of a file.

There are many version control systems available, one of which is `cvs`. Every Mac running OS X has `cvs` and Xcode supports `cvs`, which is why I chose `cvs` as the topic of this chapter.

### NOTE

The `cvs` version that ships with Mac OS X 10.4 may cause problems running `cvs` commands, especially when using a `cvs` wrappers file. Type `ocvs` instead of `cvs` to run your `cvs` commands with the old version of `cvs`.

## What You Must Do From the Command Line

Xcode has built-in support for `cvs`, and you can perform most version control tasks in Xcode, which you will see later in the chapter. But there's more work involved in using `cvs` with Xcode than launching Xcode. Before you can use `cvs` with Xcode, you must perform the following tasks:

- Create a repository, which is where `cvs` stores the files you place under version control.
- Add your project's files to the repository.
- Check the files out of the repository. Xcode can perform version control tasks only on checked out files.

There are two kinds of repositories: local and remote. A *local repository* resides on your computer. A *remote repository* resides on another computer. The other computer can be a computer you're connected to on a network or it can be a website you access from an Internet browser.

## Setting the `$CVSROOT` Environment Variable

`cvs` uses an environment variable, `$CVSROOT`, which stores the location of your repository. Before creating the repository, you can set `$CVSROOT` with a little Unix shell script programming.

Mac OS X has two possible default shells for writing shell scripts, `tcsh` and `bash`. Apple changed the default shell from `tcsh` to `bash` in Mac OS X 10.3. However, if you upgraded your Mac to 10.3 from an earlier version of OS X, the operating system kept `tcsh` as the default shell.

The previous paragraph may have left you wondering what the default shell is on your Mac. When you open a shell window, the default shell's name appears in the window's title bar. To change the default shell, choose Terminal > Preferences. Enter the path to the default shell in the text box in the middle of the preferences panel.

If you use `tcsh`, set `$CVSROOT` by using the `setenv` command.

```
setenv CVSROOT [Directory]
```

*Directory* is where you're going to place the repository when you create it. If you want to store your repository in a directory called `CodeRepository` on your hard drive, you would use the command.

```
setenv CVSROOT /CodeRepository
```

If you use `bash`, set `$CVSROOT` by giving it a directory name, then using the `export` command. The following command sets `$CVSROOT` to a directory called `CodeRepository` on your hard drive:

```
export CVSROOT=/CodeRepository
```

If writing Unix shell scripts terrifies you, run your `cvs` commands with the `-d` option. The `-d` option overrides `$CVSROOT` with a directory you supply. This directory is where the repository resides.

```
cvs -d [Directory]
```

## Creating a Repository

Creating a repository is easy; use the `cvs init` command.

```
cvs init
```

Assuming you set the `$CVSROOT` variable like I explained last section, the listing above is all you need to create a repository on your Mac. If you didn't set `$CVSROOT`, you must run `cvs` with the `-d` option to override the `$CVSROOT` environment variable. To create a repository on your hard drive with the directory name `CodeRepository`, use the command.

```
cvs -d /CodeRepository init
```

Notice how you specify the `-d` option and the repository location before the `init` command. If you don't set the `$CVSROOT` environment variable, you'll be specifying the `-d` option with the repository location before every `cvs` command you run.

Some of you may be wondering how many repositories you need to create on your machine. One repository? One repository for each project? In most cases the best solution is to have one repository and place all your projects in the one repository. Having one repository keeps things simple; you always know which repository is the current one. Even though one repository works best, there's nothing stopping you from creating multiple repositories on your computer.

## Letting Other People Access the Repository

Creating a remote repository doesn't do much good unless other people can access it. To let other people access the repository you created, go to the computer hosting the repository and perform the following steps:

- 1) Choose Apple > System Preferences.
- 2) Choose Sharing from the System Preferences panel.
- 3) Select the Remote Login checkbox to activate remote login.

Activating remote login allows other computers to access your computer so they can check files out of the repository. The other computers use Secure Shell (SSH) to access your computer. When you activate remote login, everyone connected to your computer can check files in and out of the repository, which may or may not be what you want.

If you want to give repository access to only certain people on your network, add a file called `writers` to the `CVSROOT` directory in the repository. Add all the users you want to be able to write to the repository to the `writers` file. Everybody else has read-only access. Using a `writers` file is a good solution if you have a repository on a website. The `writers` file prevents the general public from screwing up your repository while allowing them to download the files to their computers.

## Dealing with Binary and Bundled Files

The people who developed `cvs` designed it to compare text files. They also designed `cvs` to treat each file as a single file. But Mac OS X has binary files and bundled files. A binary file is a file that contains more than plain text. Examples of binary files are graphics files (JPEG, TIFF, and GIF), PDF files, and Microsoft Word files. Interface Builder nib files are bundled, which means there's many files wrapped inside the nib file.

### NOTE

You should save your Cocoa nib files as XML files if you're going to place them under version control. Open the nib file in Interface Builder and click the Nib tab. Select the Use text archive format checkbox to save the nib file as an XML file.

Before you can add binary and bundled files to your repository, you must tell `cvs` the file types that are binary and bundled. If you add binary and bundled files to a repository without telling `cvs` about them, `cvs` will mangle the files, treating binary files as text files and treating bundled files as single files.

The `cvswrappers` file tells `cvs` how to handle binary and bundled files. Initially `cvswrappers` is empty so you must modify it to tell `cvs` how to handle your program's binary and bundled files. If you go to your repository, you will see a `CVSROOT` directory. Inside the `CVSROOT` directory is the `cvswrappers` file. You must check the `cvswrappers` file out so you can modify it. To check out the `cvswrappers` file, use the `cvs checkout` command. Go to the repository directory and type the following command:

```
cvs checkout CVSROOT
```

Checking out the `CVSROOT` directory checks out all of the files in the `CVSROOT` directory, including `cvswrappers`. If you try to check out only `cvswrappers`, `cvs` reports an error message.

What you should put in the `cvswrappers` file? Fortunately, Apple includes a sample `cvswrappers` file with the Xcode Tools that covers most file types you'll need. You can open the `cvswrappers` file at `/Developer/Tools/cvswrappers`. Paste the contents of Apple's `cvswrappers` file to your `cvswrappers` file. Add any other file types you need and save your file. Now you're ready to check the `cvswrappers` file in the repository. Move to the `CVSROOT` directory and run the `cvs commit` command. The `cvs commit` command checks a file in the repository. The `cvs commit` command takes the form.

```
cvs commit -m [Message][Filename]
```

The `-m` option tells `cvs` you're going to be supplying a message. *Message* is a string you supply to `cvs` detailing the changes you made to the file, as you can see in the following example:

```
cvcs commit -m "Adding support for binary and bundled files using Apple
example." cvswrappers
```

Notice that you commit `cvswrappers`, not the `CVSROOT` directory. Now you can add binary and bundled files to your `cvcs` repository with no fear.

## Accessing a Remote Repository Using ssh

Reaching a local repository is easy; move to the directory where the repository resides. There's a little more work required to reach a remote repository. Accessing a remote repository requires at least two steps.

- Tell `cvcs` how to access the remote repository.
- Set the `$CVSROOT` environment variable to the remote repository.

In addition you can generate a key identifying yourself and send it to the computer hosting the repository. This step is not mandatory, but it makes accessing a remote repository easier.

### Telling `cvcs` to Use `ssh` for Remote Access

To access a remote repository you must tell `cvcs` how you're going to access the repository. In `cvcs` this involves setting the `CVS_RSH` environment variable on your computer. What do you set `CVS_RSH` to? The best way to access a remote repository is with `ssh`; it's the most secure way to access the repository and is also the easiest to set up. Because you're accessing the repository with `ssh`, you must set `CVS_RSH` to `ssh`. Use the `setenv` command to set `CVS_RSH` to `ssh`.

```
setenv CVS_RSH ssh
```

Using the `setenv` command works well if you don't check out many files. If you find yourself checking out files often, using `setenv` every time you want to check out a file can be a pain. To ease the pain, you can set the `CVS_RSH` environment variable in your environment variable property list file, which contains a list of environment variables and their values that is set automatically when you login to your Mac. You can find the environment variable property list file at `~/MacOSX/environment.plist`. The `environment.plist` file may not be visible in the Finder. Do a search on your hard drive for `environment.plist` to find it. To set the `CVS_RSH` environment variable in your `environment.plist` file:

- 1) Double-click the `environment.plist` file to open it in the Property List Editor program.
- 2) Click the disclosure triangle next to Root.
- 3) Click the New Child button to create a new key value pair.
- 4) Change the name of the item to `CVS_RSH`.
- 5) Change the value to `/usr/bin/ssh`.
- 6) Save the file.

Now when you login to your computer, the operating system automatically sets the `CVS_RSH` environment variable to `ssh`.

### Setting \$CVSROOT to a Remote Repository

When setting the \$CVSROOT environment variable for a local repository, you set it to the directory containing the repository. The location of a remote repository is more difficult to name, taking the form.

```
:ext:Username@MachineName:[Repository Directory]
```

*Username* is your Mac OS X user name. *MachineName* is the name of the computer hosting the repository. If you're on a local area network, the name of the computer will be something like `ComputerName.local`. To find the actual name, choose Go > Network from the Finder. Select Local to see the names of the computers you're connected to.

If you're accessing a cvs repository on the World Wide Web, the machine name takes the form of an Internet domain name like `www.CompanyName.com`. Many online cvs repositories take the form `cvs.CompanyName.com`. *Repository Directory* is the directory where the repository resides on the computer hosting the repository.

Let's look at two examples of setting \$CVSROOT to point to a remote repository. The first example uses a local area network. Andy wants to access the repository, which resides on the computer named `ComapnyServer.local`. The repository is in the directory `/CodeRepository`. In `tcsh`, Andy would set \$CVSROOT with the following command:

```
setenv CVSROOT :ext:andy@CompanyServer.local:/CodeRepository
```

In `bash` Andy would use the following command to set \$CVSROOT:

```
export CVSROOT=:ext:andy@CompanyServer.local:/CodeRepository
```

In the second example, Andy wants to access the repository from his company website at `cvs.example.com`. The repository resides in the directory `/CodeRepository` on the website. He would set \$CVSROOT in `tcsh` using the following command:

```
setenv CVSROOT :ext:andy@cvs.example.com:/CodeRepository
```

Andy would set \$CVSROOT in `bash` using the following command:

```
export CVSROOT=:ext:andy@cvs.example.com:/CodeRepository
```

After setting \$CVSROOT you can access a remote repository, but you must enter your password every time you access the repository. To avoid entering your password for every cvs command you submit to the repository, `ssh` lets you create public and private keys that let the remote computer know your identity.

### Generating Keys

Use the `ssh-keygen` command to generate the public and private keys. What you supply to the `ssh-keygen` command depends on the version of `ssh` you're using. Most of you have version 2, the latest one. In version 2, use the following command to generate the keys:

```
ssh-keygen -t dsa
```

If you have version 1 of `ssh`, use the following command to generate the keys:

```
ssh-keygen -t rsa1
```

When you run the `ssh-keygen` command, `ssh` asks you for a file name to save the keys. `ssh` provides a default file in parentheses with the path `/Users/YourUsername/.ssh/Filename`. Press the Return key to use the default file. Next, `ssh` asks you for a password. Be careful typing your password. The shell window provides no feedback when typing the password so it looks like you're not typing anything, even though you are typing the password. After providing a file location and password, `ssh` generates two keys. The public key has the extension `.pub`, and the private key has no extension. In `ssh` version 2 the key name is `id_dsa`. In version 1, the key name is `identity`. Keep the private key on your computer and send the public key to the computer hosting the repository.

## Moving Your Key to the Repository

Generating the keys does no good unless the computer hosting the repository knows about them. You must place your public key on the host computer so the host can identify you without requiring a password when you login. The public key goes into the file `authorized_keys2` inside the `.ssh` directory on the host computer. The `authorized_keys2` file contains the keys of all the people authorized to access the repository without a password.

Assuming there's no `.ssh` directory and `authorized_keys2` file on the host computer, you must run `ssh` to login and use the `mkdir` command to create the `.ssh` directory. If Andy wanted to create the `.ssh` directory on his company's server, `CompanyServer.local`, he would use the following commands:

```
ssh andy@CompanyServer.local
mkdir .ssh
```

If Andy wanted to create the `.ssh` directory on his company's website, `cvs.example.com`, he would use the following commands:

```
ssh andy@cvs.example.com
mkdir .ssh
```

To copy the public key to the host computer, use the `scp` (secure copy) command. `scp` allows you to copy a file from your computer to another computer. For our purposes, `scp` takes the form.

```
scp Filename Username@MachineName
```

If Andy wanted to copy his public key to the company server, he would use the following command:

```
scp ~/.ssh/id_dsa.pub andy@CompanyServer.local
```

To copy his public key to the company website, he would use the following command:

```
scp ~/.ssh/id_dsa.pub andy@cvs.example.com
```

After copying the public key file to the host computer, you must add the key to the `authorized_keys2` file. The `cat` command appends a file to the contents of another file. Use the `cat` command to append the public key file, `id_dsa.pub`, to the list of keys in `authorized_keys2`. After calling `cat` you can remove the copy of `id_dsa.pub` on the host computer by using the `rm` command, as you can see in the following example:

```
cat id_dsa.pub > ~/.ssh/authorized_keys2
rm id_dsa.pub
```

## Adding a Project to the Repository

The easiest way to get your code into a `cvs` repository is to move an existing Xcode project. Trust me, it beats adding each source code file to the repository individually. If you haven't done so already, create an Xcode project for the program you want to place under version control. Don't worry about adding all the files you want to add to the project; you can add them to the repository from Xcode later. What's important is creating the Xcode project. Move to your project's directory and run the `cvs import` command.

```
cvs import -m [Message] [Destination] [Vendor Tag] [Release Tag]
```

I strongly recommend you use the `-m` option and supply a message. If you don't use the `-m` option and supply a message, `cvs` prompts you to type in a log message. It's not clear how you finish the message so `cvs` can import the files. You'll save yourself a lot of aggravation by using the `-m` option and supplying a message.

*Destination* is the directory where your files will reside in the repository. This directory must be a subdirectory of the repository. *Vendor Tag* and *Release Tag* are arguments `cvs` requires. It doesn't matter what you put in these tags. I would suggest using your name or your company's name as the vendor tag and use any string you wish as the release tag.

```
cvs import -m "Importing project" MyProject/Files mark first
```

Running `cvs import` like the listing above creates a subdirectory in your repository titled `MyProject`. Inside the `MyProject` folder is a folder with the name `Files`. The `Files` folder holds all the files in your project. When you check out a source code file to work on it, you will be checking it out from the `Files` folder.

## Checking Files Out of the Repository

Before you check files out of the repository, make a backup copy of your project folder. If something goes wrong during the checkout, you have an original version of your files you can go back to.

To check files out of the repository, use the `cvs checkout` command. To check out all of the files in your project, go to the directory above your project folder. Supply your project folder's name to `cvs checkout`.

```
cvs checkout MyProjectFolderName
```

If you go to your project folder, you will see two new folders. The first folder is the `CVS` folder, which contains information about the repository and its entries. The name of the second folder depends on the destination you supplied to `cvs` when you ran the `cvs import` command. If you used my naming convention, the folder will have the name `Files`. If you look inside the `Files` folder, you will see it has all your project's files and folders. These files and folders are your working copies that are under version control. The original copies are in the repository.

After checking out the files, you have two copies of every file in your project. One copy is in the project folder and the other copy is in the `Files` folder inside your project folder. Which copy should you use? Use the copy in the `Files` folder. The `Files` folder copy is under version control. The copy in the project folder isn't. You can remove the copy in the project folder; you don't need it anymore. Keeping the project folder copy serves only one purpose: to confuse you.

To check out one file from your project, go to your project folder and run `cvs checkout` with the file's name.

```
cvs checkout Filename
```



## Using cvs in Xcode

If you made it to this point in the chapter, I congratulate you. The dirty work is out of the way. Once you import your project in the repository and check out the project's files, you can leave the command line and move to Xcode. From Xcode you can perform the most common version control tasks, which I cover in the rest of the chapter.

### Turning on Version Control

The first version control task to perform in Xcode is to turn on version control.

- 1) Select the project file from the Groups and Files list.
- 2) Click the Info button to show the project file's information panel.
- 3) Click the General tab.
- 4) Select the Enable SCM checkbox.
- 5) Choose CVS from the SCM System pop-up menu.

If you're connecting to a remote repository with `ssh`, you must tell Xcode about the connection.

- 6) Click the Edit button next to the SCM System pop-up menu. A dialog box opens.
- 7) Select the Use `ssh` instead of `rsh` for external connections checkbox.
- 8) Click the OK button.

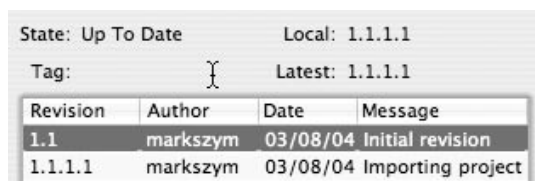
### Did the Command-Line Checkout Succeed?

After turning on version control, you want to see if the checkout worked. It's aggravating to change a file and send the changes to the repository only to find out the file hasn't been checked out. You can see if your checkout succeeded by viewing the file's Source Code Management (SCM) information panel. To view the SCM information for a file, you must perform the following steps:

- 1) Select the source code file from the project window.
- 2) Click the Info button to open the file's information panel.
- 3) Click the SCM tab.

If everything worked properly, there will be a listing for the file with a version number, author and date in the SCM information panel. Figure 11.1 shows an example of the information panel after an initial import. The message for the version is the message you entered when you ran the `cvs import` command. If the checkout did not succeed, you must go back to the command line and run `cvs import` again.

Now that you have version control working for your project, you can go about the normal business of development: writing code, compiling it, testing it, and debugging it.



State: Up To Date		Local: 1.1.1.1	
Tag:		Latest: 1.1.1.1	
Revision	Author	Date	Message
1.1	markszym	03/08/04	Initial revision
1.1.1.1	markszym	03/08/04	Importing project

**Figure 11.1**

The SCM information panel for a file in a project that has been imported into `cvs`.

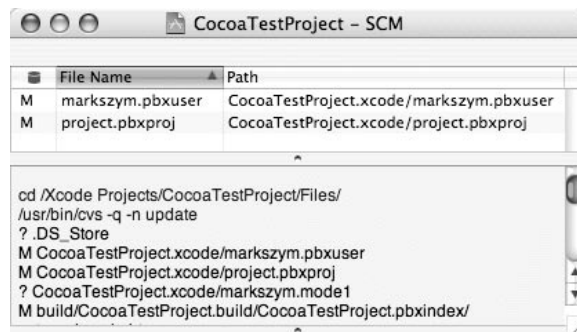
## Seeing Which Files Aren't Up To Date

Initially the files in your project match the files in the repository. As you work on your project, editing source files and adding files to the project, some files no longer match. These files are the ones you must update in the repository. To see the list of files that don't match the saved version in the repository, select SCM from the Groups and Files list. The project window lists the files that don't match. The left column has a one-character code representing the file's version control status. Table 11.1 lists the possible codes.

If you want to see the same information in a separate window, choose **SCM > SCM**. Showing the SCM information in a separate window provides a bonus, which you can see in Figure 11.2. The bonus is a drawer that lists the SCM status of files in your project's folder that aren't part of your project in addition to the files the window shows at the top of the window. The drawer also tracks recent version control activity you've made. Initially the drawer isn't visible; you must drag the splitter bar at the bottom of the window to see the drawer.

**Table 11.1 Version Control Status Codes**

Code	Description
?	The file is not in the repository. When you add a file to your project, it won't be in the repository.
-	The file is in a folder that's not in the repository. You must add the folder to the repository from the command line.
M	You've modified the file. Choose <b>SCM &gt; Commit Changes</b> to save your changes to the repository.
A	To be added. The file will be added to the repository the next time you choose <b>SCM &gt; Commit Changes</b> .
R	To be removed. The file will be removed from the repository the next time you choose <b>SCM &gt; Commit Changes</b> .
C	The changes you made to the file may conflict with the changes the latest version made. You would get this code if another programmer modified a file while you checked that file out.
U	The version of the file you're using is older than the latest version in the repository.



**Figure 11.2**

The Xcode SCM window with the drawer open.

## Committing Changes You Made

You've made changes to one of your source code files. It could be new code, a bug fix, a speed boost, or taking some ugly code that somehow works and cleaning it up. You tested the changes you made, and everything works perfectly. At this point you want to send the changes you made to the file to the repository, creating a new version of the file in the repository.

When you send your changes to the repository, Xcode prompts you for a message where you describe the changes you made. Before sending the changes, you may want to look at the changes you made to the file so you can type a meaningful message when prompted by Xcode. To see the changes you made to a file, choose **SCM > Compare with > Latest**. One annoying bug in FileMerge, the program that compares the files, is it occasionally treats the entire file as one difference. When FileMerge treats the entire file as one difference, you can't tell the differences between your version of the file and the version stored in the repository.

When you're ready to send your changes to the repository, choose **SCM > Commit Changes**. Xcode prompts you to type in a message describing the changes you've made. After typing the message, click the Commit button to send the changes. Now there's a new version of the file in the cvs repository.

## Discarding Changes You Made to a File

Suppose you made some changes in a file and found out the changes didn't work. How do you go back to where you were before you made the changes? Choose **SCM > Discard Changes**. Xcode erases all the changes you made and takes you back to the last version you sent to the repository. Discarding changes is the solution when you save a file and wish you hadn't.

## Adding Files to the Repository

If you create an Xcode project and check the project's files out, you can add files to the project and add them to the cvs repository from Xcode. To add a file to the repository:

- 1) Select the file you want to add from the project window.
- 2) Choose **SCM > Add to Repository**. The file you want to add has the status A, which means it's ready to be added to the repository.
- 3) Choose **SCM > Commit Changes** to add the file to the repository.

## Removing Files from the Repository

Although adding files to a repository is more common than removing them, you can remove files from a cvs repository from Xcode. To remove a file from the repository:

- 1) Select the file you want to remove from the Groups and Files list.
- 2) Press the Delete key.
- 3) An alert opens asking if you want to remove the file from disk or just the reference to the file from the project. Removing the reference is safer; you can drag the file to the Trash if you want to delete it.
- 4) A second alert appears asking if you want to remove the file from the SCM repository. Click the Remove button.
- 5) Select **SCM** from the Groups and Files pane. The file you want to remove should have an R next to it saying it's ready to be removed.
- 6) Select **SCM > Commit Changes** to remove the file from the repository.

## Seeing a File's SCM Information

After saving several versions of a file, the file's SCM information panel provides additional usefulness. Selecting a version from the panel fills the bottom area of the panel with information about that version of the file, including.

- The file name.
- The version number.
- Who checked the version into the repository.
- When the person checked the version into the repository.
- A message describing the changes made in this version.

## Comparing Two Versions of a File

If you select two versions of a file in the SCM information panel, the Compare and Diff buttons become enabled. Clicking the Compare button shows the differences between the two versions in the FileMerge application. Clicking the Diff buttons opens a window in Xcode. In the window are the two versions of the file you're comparing side by side. Seeing the differences using the Diff button is more difficult. A changed line of code is indented a few spaces in the right hand version of the file, which isn't as noticeable as FileMerge's highlighting of the differences.

## Reverting to an Old Version of a File

If you select an old version of a file in the SCM information panel, the Update button becomes enabled. Clicking the Update button sets the current version of the file to the selected version.

If you select the latest version of a file, and you've made changes to the file recently, the text of the Update button changes to Discard. Clicking the Discard button takes you back to the last version you saved to the repository, erasing the changes you made.

## Viewing cvs Annotations

When multiple people make multiple changes to a file, it's nice to know who changed what in the file. Annotations perform this vital task, telling you the following information for each line in a source code file:

- The version of the file that modified the line of code.
- Who modified the line.
- When that person modified the line.

You must have at least two versions of a file in the repository before you can view the file's annotations. To view the annotations, select the file from the project window and choose **SCM > Get Annotations for > Latest**. To see annotations for an older version of the file, choose **SCM > Get Annotations for > Revision**. A dialog appears with a list of all the versions for the file. Select the version you want and click the Annotate button.

# Chapter 12

## Java Tools

One of Apple's goals in Mac OS X was to make it the premier Java platform. Many of the Java developer tools are integrated with Xcode. This chapter covers the other Java tools that ship with the Xcode Tools.

### Jar Bundler

The Jar Bundler program takes a Java application and creates a Mac OS X application bundle users can launch from the Finder. If you don't create an application bundle with Jar Bundler, users must run the Terminal program and use the `java` command-line tool to launch your Java application. Use Jar Bundler to create an application that looks like any other Mac OS X application and keep your code compatible with other operating systems.

#### NOTE

Xcode creates application bundles for Cocoa Java applications so you don't have to run Jar Bundler on a Cocoa Java application.

When you launch Jar Bundler, a window like Figure 12.1 opens. The window has three tabs.

- Build Information, which lets you determine how the application behaves in Mac OS X.
- Classpath and Files, which lets you add files to the bundle.
- Properties, which lets you set properties for the application's property list file.

After setting the information in each tab, click the Create Application button to create the bundled application.

### Setting Build Information

The build information panel is the initial view in Jar Bundler, which you can see in Figure 12.1. Mac OS X applications and Java Swing applications have subtle differences in their behavior. Use the build information panel to settle the differences, specifying when your program should act like a Mac OS X application and when it should act like a Swing application. The build information panel also lets you set the Java properties of the application bundle Jar Bundler creates.



**Figure 12.1**

Initial JarBundler window.

As you can see in Figure 12.1, there are a lot of controls in the pane. Rather than having one massive section detailing all the controls, I've given each control its own section.

## Options for Main

Every Java program requires a `main()` function in one of the program's classes. The `main()` function is the starting point of the program. You must tell Jar Bundler which class in your program has the `main()` function so Jar Bundler can create the bundle. Click the Choose button, then select the file containing your program's `main()` function.

In a Java program you can supply arguments to the `main()` function. If you have runtime arguments in your program, supply them in the Arguments to Main text field. Only programs that run from the command line use runtime arguments so you usually don't have to worry about the Arguments to Main text field.

## Custom Icon

If you have a custom icon for your application, click the Choose Icon button. Select the icon file from the dialog box, and the icon appears above the Choose Icon button. If you don't have a custom icon, Jar Bundler uses the generic Java application icon that shows a cup of coffee.

## JVM Version

The JVM Version pop-up menu lets you select the version of the Java virtual machine your program will use. Choosing the Java virtual machine version is the most important decision you'll make with Jar Bundler because the Java virtual machine version determines the versions of Mac OS X that can run your program. Mac OS X 10.2 ships with Java virtual machine version 1.3.1, and Mac OS X 10.3 ships with version 1.4.1. Requiring version 1.4.1 in your program means users running Mac OS X 10.2 would have to download Java virtual machine version 1.4.1 to be able to run the program. You'll see three types of versions in the pop-up menu.

- A specific version, such as 1.4.1.
- The highest version of a specific Java release. It appears in the pop-up menu with an asterisk, such as 1.4\*. The 1.4\* says to use the highest 1.4 version of the Java virtual machine. If the highest available 1.4 version is 1.4.2, the 1.4.2 version will be used. However, if there were a 1.5 version available, it would not be used.
- The highest possible version of the Java virtual machine. It appears in the pop-up menu with a plus sign, such as 1.4+. The 1.4+ says to use the most recent version of the Java virtual machine, but the version must be at least 1.4.

Which Java virtual machine version you should choose? Using one of the versions with a plus sign is best in general cases because your application will be able to use future versions of the Java virtual machine and run on future versions of Mac OS X. Use the earliest Java virtual machine version you can so your program can run on the widest possible versions of Mac OS X.

If your program uses a feature available in only certain versions of the Java virtual machine, then use a version with an asterisk or a specific version. Version 1.4 of the Java virtual machine does not take advantage of hardware acceleration. If you want your program to use the user's graphics card to draw windows, choose version 1.3\*. Choosing 1.3+ would make your program use version 1.4 on Macs with that version, and hardware acceleration would not be used. Don't worry about users of newer versions of Mac OS X not being able to run Java programs that use virtual machine version 1.3. Java virtual machine versions are backwards-compatible, meaning a newer version of the Java virtual machine can run programs written for earlier versions. Java virtual machine 1.4 and higher versions can run programs written for Java virtual machine version 1.3.

## Use Macintosh Menu Bar

The Use Macintosh Menu Bar checkbox determines the appearance of the menu bar in your program. Mac OS X programs place the menu bar at the top of the screen. Java programs using the AWT or Swing frameworks place a menu bar in each window. Selecting the Use Macintosh Menu Bar checkbox means your program will have one menu bar at the top of the screen. Otherwise your program will have a menu bar in each window.

## Anti-alias Text and Graphics

When drawing lines on a computer, the computer fills the pixels on the screen with color. For lines that aren't vertical or horizontal, lines can look jagged, which you can see for yourself by taking a sheet of graph paper and drawing lines by filling squares on the graph paper. The jagged edges that occur when using pixels to draw lines are a sign of *aliasing*. *Antialiasing* smooths the lines so they look better on the screen. Antialiased text and graphics look better, but run slower. Looks versus speed is the decision you must make with the Anti-alias Text and Anti-alias Graphics checkboxes.

## Growbox Intrudes

The Growbox Intrudes checkbox lets you specify whether the resize control intrudes on the window's content area. If you do not select the checkbox, a white bar appears at the bottom of windows that have a resize control.

## Disable .app Package Navigation

The Disable .app Package Navigation checkbox determines whether or not users can access the contents of a bundle through the AWT file dialogs. Bundles appear to the user as a single file, but there are directories beneath the surface. If you don't select the Disable .app Package Navigation checkbox, users will be able to navigate the directories beneath the surface. You normally should select this checkbox. The whole point of bundles is to reduce complexity for the user. Allowing users to navigate the bundle increases the complexity.

## Live Resizing

Selecting the Live resizing checkbox tells Jar Bundler turns on live window resizing. What is live window resizing? As the user drags the mouse to resize the window, the operating system draws the contents of the window. Most windows in Mac OS X programs have live window resizing. With live resizing off, the operating system draws an outline showing the size of the resized window as the user resizes. The operating system doesn't draw the window's contents until the user stops resizing. To see a window without live resizing, run an old Mac program that runs in the Classic environment.

## Enable Hardware Acceleration

Apple introduced Quartz Extreme in Mac OS X 10.2. In Quartz Extreme the operating system uses the graphics card to draw windows and their contents. Using the graphics card makes window scrolling, moving, and resizing faster. Selecting the Enable Hardware Acceleration checkbox tells Jar Bundler to use Quartz Extreme in your Java program.

## Smaller Tab Sizes

The tabs the Smaller Tab Sizes checkbox refer to are the tabs of tabbed windows, such as the Jar Bundler window. Selecting the Smaller Tab Sizes checkbox shrinks the font size of text in tabs, which shrinks the tab.

## Adding Files to the Bundle

In the build information panel you set the main class for the application. Larger Java applications have multiple `jar` files and may use class libraries. Some applications may include graphics, sound, and video files. The classpath and files panel, shown in Figure 12.2, lets you add files to the application bundle.

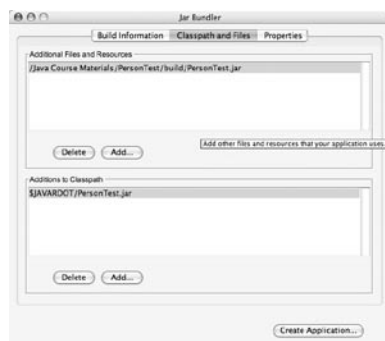
Adding files to the bundle is easy. Click the Add button to open a dialog box asking you for a file to add. When you add a file to the bundle, Jar Bundler adds the file's path to the classpath list. You can add more paths to the classpath list by clicking the Add button in the Additions to Classpath group box.

## Setting Properties

The properties panel, shown in Figure 12.3, lets you set entries for the application's property list file. Looking at Figure 12.3, you can see there are a lot of controls in the panel. Rather than having one massive section detailing all the controls, I've given each control its own section.

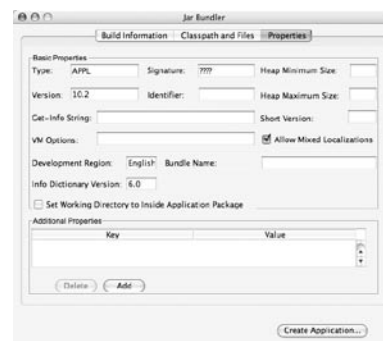
## Type

The Type text field specifies the type of package Jar Bundler creates. The package type is a four-character code. Jar Bundler supplies the code `APPL`, which is the code for an application bundle. Since Jar Bundler is a program to create application bundles, the `APPL` code is the code you want to use.



**Figure 12.2**

Classpath and files panel.



**Figure 12.3**

Properties panel.



## Signature

The signature is a four-character code that uniquely identifies your application. I strongly recommend using the Identifier text field to identify your program instead of using the Signature text field. If you use a signature, you must register it with Apple to ensure the signature doesn't conflict with another program. It's much simpler to leave the four question marks as the signature and use the Identifier text field.

## Heap Size

The Heap Minimum Size and Heap Maximum Size text fields specify the size of the Java virtual machine heap. The default minimum size is 2 MB, and the default maximum size is 64 MB. The minimum and maximum sizes must be a multiple of 1024 bytes, 1KB. The maximum size has a limit of 2GB. The larger the heap size, the less often the Java virtual machine has to collect garbage, but the garbage collection takes longer because there's more garbage to collect.

## Version

The Version text field is where you enter your application's version information. Enter your application's version number and copyright information.

## Identifier

The Identifier text field uniquely identifies your application. The identifier takes the form.

```
com.CompanyName.ApplicationName
```

Jar Bundler has the following identifier:

```
com.apple.JarBundler
```

## Get-Info String

The Get-Info String text field contains the text that appears when the user gets information about your program by choosing File > Get Info from the Finder.

## Short Version

The Short Version text field is where you place the version number for your application.

## VM Options

VM Options are options you can supply dealing with the Java virtual machine. Most of the VM options can be set in other parts of Jar Bundler so you normally don't have to set any options in the VM Options text field. The options you would be most likely to use deal with Java garbage collection and threading. Refer to the developer documentation at Documentation > Java > Reference > Java Virtual Machine Options for a complete list of VM options.

VM options look similar to `gcc` compiler flags. Most of the options start with `-X`. The following option turns off garbage collection:

```
-Xnoclassgc
```

## **Allow Mixed Localizations**

The Allow Mixed Localizations checkbox specifies whether different language versions of your program can exist in the same bundle.

## **Development Region**

The Development Region text field specifies the native language for the application. If you live in North America, Jar Bundler places English in the text field for you.

## **Bundle Name**

The Bundle Name text field shows the bundle's name, which is the name that appears in the Application menu when people run your program.

## **Info Dictionary Version**

The Info Dictionary Version text field contains the version number of the information property list file. You shouldn't need to change the version Jar Bundler supplies.

## **Set Working Directory Inside Application Package**

The Set Working Directory to Inside Application Package checkbox specifies whether the initial working directory should be inside the bundle. You shouldn't have to worry about this checkbox unless you add non-Java files (data, graphics, and sound files) to the bundle. If you add non-Java files to the bundle, selecting the checkbox makes loading the non-Java files easier.

## **Additional Properties**

The Additional Properties table lets you manually add properties to the bundle's property list. Click the Add button to add a property, then fill the Key and Value fields for the property.

A common case of having to add properties to the property list is when you have an application containing online help files. If your application has online help files, you must set the property `CFBundleHelpBookFolder`, which contains the folder where the help files reside. In Jar Bundler terminology, the key is `CFBundleHelpBookFolder`, and the value is the folder name containing the help files. You can see a complete list of keys in the developer documentation at [ADC Reference Library > documentation > Mac OS X > Runtime Configuration > Property List Key Reference](#).

## JavaBrowser

JavaBrowser is a documentation viewer for Java class libraries. Use it to look up information on Sun's and Apple's Java classes. Looking at class documentation may not seem like a big deal since you can view Java class information in Xcode, but JavaBrowser has additional features. You can view the source code for many of Sun's classes. You can even add your own Java classes to JavaBrowser.

### The Browser Window

When you run JavaBrowser, a window like Figure 12.4 opens. The window consists of the following components:

- Class browser
- Summary area
- Documentation area
- Display buttons

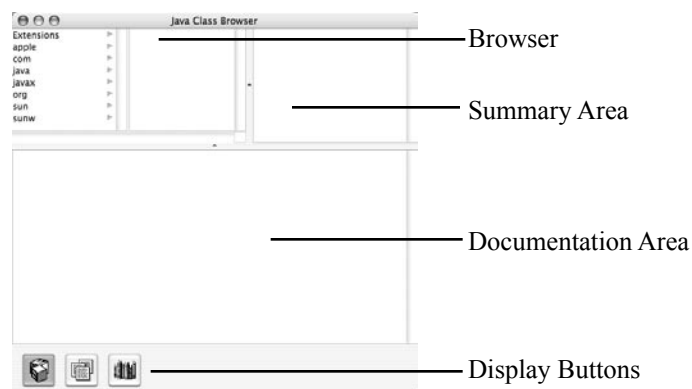
Use the class browser to navigate the Java classes. If a class in the browser has a triangle next to it, the class has one or more subclasses. Select the class to see its subclasses.

When you reach a class in the browser, the summary area lists the following information for the class:

- Fields, the class' data members.
- Constructors. A *constructor* is a function your program calls when it creates an object of the class.
- Methods, the class' member functions.

If you're looking at the source code in the browser, selecting a field, constructor, or method from the summary area takes you to what you selected in the documentation area.

The documentation area displays the documentation. What appears in the documentation area depends on the display button you choose. You may end up seeing the class' description, source code, or HTML documentation.



**Figure 12.4**

JavaBrowser browser window.

## Viewing a Class's Description

The initial view is the class's description, which you can always bring up by clicking the left display button. The class description looks like a header file in C++ or Objective C, listing the class's fields (data members), constructors, and methods.

## Viewing a Class's Source Code File

Clicking the middle display button opens the class' source code file in the documentation area of the browser window. Not every class makes its source code available; you can't view the source code for any of Apple's classes. You can view the source code for most of Sun's classes; look through the `java`, `javax`, and `org` packages to see classes brimming with source code.

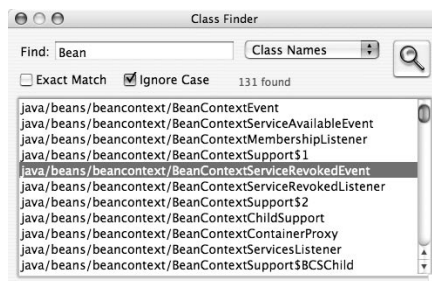
## Viewing a Class's Documentation

Clicking the right display button opens the class's HTML documentation in the documentation area of the browser window. You may not be able to view the documentation initially. When you install the Xcode Tools, the installer leaves the Java documentation compressed, meaning you can't view the documentation files. If you can't view the documentation, a message appears in the browser window with instructions on how to decompress the documentation.

## Searching for Information

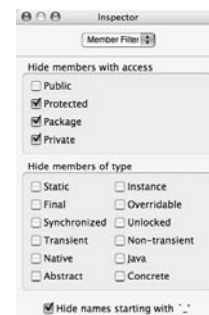
If you spend some time browsing the Java classes, you'll notice there are lots of classes, many of which you won't be using in your Java programs. The amount of information is overwhelming, especially when you're trying to find a specific piece of information. The class finder lets you search for class names, fields, and methods.

Choose Tools > Show Class Finder to open the class finder. Figure 12.5 shows the initial class finder. Type what you're looking for in the text field. The pop-up menu tells JavaBrowser what to search for. You can search for class names, field names, method names, or all three. Click the button with the magnifying glass, and the class finder displays the information you were looking for. Selecting an entry from the class finder displays that entry's information in the browser window.



**Figure 12.5**

Class Finder dialog.



**Figure 12.6**

Inspector panel.

## Filtering Class Information

The inspector panel lets you filter the information that appears in the browser window. Choose Tools > Inspector to open the inspector panel, which you can see in Figure 12.6. The most interesting part of the inspector window is the member filter. The member filter tells JavaBrowser the fields and methods to display in the browser window. JavaBrowser can filter class members by their access level and their type.

Table 12.1 lists the Java access levels. Initially the browser window displays only public members of the classes. Displaying only public members makes sense because the class browser initially shows classes you can't modify anyway. If you're interested in learning about the details of the Sun and Apple Java classes, deselect the boxes in the group Hide Members with Access. Table 12.2 lists the member types you can filter in JavaBrowser.

**Table 12.1 Java Access Levels**

Access Level	Description
Public	Any class can access a public member. Most methods have public access.
Protected	The class and its subclasses can access a protected member of a class.
Package	Any class in the same package can access a member of a class with package access. A package contains a group of related classes.
Private	Only the class can access a private member of the class. Fields are the members most likely to have private access.

**Table 12.2 Java Member Types**

Member Type	Description
Static	There is one instance of a static member that all objects of the class share.
Instance	Each object gets its own copy of an instance member. A member can be static or instance, but not both.
Final	Final members cannot change. Final fields represent constant values. Final methods cannot be overridden by a subclass. Final classes cannot have subclasses.
Overridable	Overridable members can change. A member can be final or overridable, but not both.
Synchronized	Multithreaded programs use synchronized methods. When an object calls a synchronized method, the program locks the object to keep other threads from calling the method.
Unlocked	Unlocked methods do not lock the object when the program calls them. A method can be synchronized or unlocked, but not both. Most methods are unlocked.
Transient	Transient variables are not part of an object's persistent state, which means the variable does not have to be saved when archiving the object.
Non-transient	Non-transient variables are part of an object's persistent state, which means the variable must be saved when archiving the object. A variable can be transient or non-transient, but not both.
Native	Native members are written in a language other than Java.
Java	Java members are written in Java. A member can be native or Java, but not both.
Abstract	Abstract classes exist to be inherited by other classes. You cannot declare an object of an abstract class. An abstract class can have abstract methods. Abstract methods are empty in the abstract class. The subclass overrides the abstract method.
Concrete	Concrete classes can stand on their own. Concrete methods contain code, unlike abstract methods, which are empty. A member can be abstract or concrete, but not both.

## Adding Your Own Classes to the Browser

Initially the browser window shows only the Java classes from Sun and Apple. You can add classes from your Java programs to the browser as well. Choose **File > Add Classes**. A dialog box opens asking you for a class file to add. Navigate to the location of the `jar` file you want to add, select it, and click the **Open** button. The classes in the `jar` file will appear in the class browser.

Initially you can see only the class description for your classes in JavaBrowser. To be able to see source code and HTML documentation, you must use either `javadoc` or `HeaderDoc`. These tools convert comments in your code into HTML files. Which tool should you use? Each tool has its own advantages. Apple has a lot of `HeaderDoc` documentation and a mailing list, making it easier for you to get started. `javadoc` is a Sun tool, and Sun uses it to generate all of its documentation. Sun using `javadoc` means there are lots of working examples to view in JavaBrowser. Look at the source code for a Sun Java class to see `javadoc` comments. By using the center and right display buttons in JavaBrowser, you can look at a `javadoc` comment and see the HTML documentation it generates. The final decision is up to you.

If you add classes to the browser and quit JavaBrowser, the classes you added won't be in the class browser the next time you run JavaBrowser. To make your classes appear in the class browser every time you run JavaBrowser, you must add your classes' file paths to JavaBrowser's path list. Open the preferences window by choosing **JavaBrowser > Preferences**. From the preferences window you can set the paths for all three display buttons. Select the appropriate display button from the preferences window toolbar and click the **Add Items** button. Choose the `jar` files you want to add to the path list — or directories for HTML documentation — and those classes will appear in the browser window every time you run JavaBrowser.

## Applet Launcher

As its title suggests, Applet Launcher is a tool to launch Java applets. It serves two purposes. The first purpose is to test your applets without having to open an Internet browser. The second purpose is to test newer applets on Java virtual machine 1.4.2.

### Launching an Applet

When you launch Applet Launcher, you see the launch window, shown in Figure 12.7. If the applet is on your Mac, click the **Open...** button to find the applet you want to launch. Click the **Launch** button to launch the applet. When you click the **Launch** button, Applet Launcher adds the applet to the History menu so you can easily relaunch it at a later date.



**Figure 12.7**

Applet Launcher launch window.

Keep in mind that applets require an HTML file to launch. If you supply your applet's jar file, Applet Launcher won't be able to launch the applet. Xcode Java applet projects include a file named `example1.html`. This file is what you must supply to Applet Launcher.

If the applet you want to launch is located on a website instead of your Mac, clicking the Open button won't work. You must type the applet's URL.

```
http://www.example.com/example1.html
```

After launching the applet, the File menu becomes the Applet menu. Use the Applet menu to stop, restart, reload, clone, and quit (close) the applet.

## Getting Applet Information

The Applet menu also lets you retrieve information about the running applet. Choose Applet > Tag to see the applet's HTML tag. Choose Applet > Info to view information about the applet and its parameters. Choose Applet > Properties to set a HTTP proxy server for the applet. You shouldn't need to set a proxy server for applets that reside on your Mac.

## Serializing Applets

Choose Applet > Save to serialize the applet. Serialization takes objects in a program and converts them into streams of data. These streams can be saved to a file and restored later.

# Chapter 13

## OpenGL Tools

One of Apple's goals with Mac OS X was to make it the premier OpenGL platform for both users and developers. To make Mac OS X the best OpenGL platform for developers, Apple includes OpenGL developer tools as part of the Xcode Tools. This chapter explains how to use these tools.

### OpenGL Profiler

OpenGL programs demand high performance to be usable, making these programs the most likely to require profiling. But if you profile an OpenGL program with Shark or one of the other performance tools, you'll notice a problem. The OpenGL function calls the program makes don't appear in the profile. Use OpenGL Profiler to profile your program's OpenGL function calls.

OpenGL Profiler works like a normal profiler, but it focuses exclusively on OpenGL function calls. For each OpenGL function call your program makes, OpenGL Profiler reports the number of times your program called the function and the amount of time spent in the function. In addition to profiling, OpenGL Profiler helps you debug your OpenGL programs. You can set breakpoints, run scripts of OpenGL commands, and view the contents of OpenGL's buffers.

### Choosing a Program to Profile

When you launch OpenGL Profiler, the main window (see Figure 13.1) opens. Your first step is choosing a program to profile. How you choose a program to profile depends on whether or not the program you want to profile is currently running. If the program is running, select the Attach to application radio button. A list of all running applications appears in the window. Select the program you want to profile from the list.

If the program you want to profile isn't running, select the Launch application radio button. A list of the programs you previously profiled appears in the window. If the program you want to profile isn't on the list, click the + button. A dialog box opens for you to choose the program to profile. If your program takes any runtime arguments, enter them in the Launch Arguments column.

If you chose to launch your application from OpenGL Profiler, the Launch Settings disclosure triangle becomes enabled. Click the disclosure triangle to use a custom pixel format, choose a graphics driver, and set environment variables.



**Figure 13.1**

OpenGL Profiler main window.



## Custom Pixel Formats

Looking at the Use custom pixel format checkbox in Figure 13.1, you might be wondering what a custom pixel format is. Pixel formats allow you to specify attributes for the draw context where OpenGL does its drawing. Do you want your program to run in a window or run fullscreen? Do you want a back buffer? Do you want your drawing hardware accelerated? What color depth do you want: 16-bit, 32-bit, or 64-bit color? How large do you want OpenGL's alpha, depth, stencil, and accumulation buffers? Pixel formats answer these questions. You have to specify a pixel format in your code for your OpenGL program to run. Using a custom pixel format lets you experiment with pixel format attributes without having to recompile your program.

To build your custom pixel format, click the Edit button. The custom pixel format window opens, which you can see in Figure 13.2. Click the Attributes button to open the drawer containing the pixel format attributes you can set. Double-click an attribute to add it to the custom pixel format. To add multiple attributes, select them from the drawer and drag them to the window. Double-click the Value column to change an attribute's value. For Boolean values, use `GL_TRUE` or 1 for true and `GL_FALSE` or 0 for false. To remove an attribute from the custom pixel format, select the attribute from the custom pixel format window and press the Delete key. Clicking the Clear button removes all the attributes from the pixel format. Close the window when you're finished.

## Choosing a Graphics Card Driver

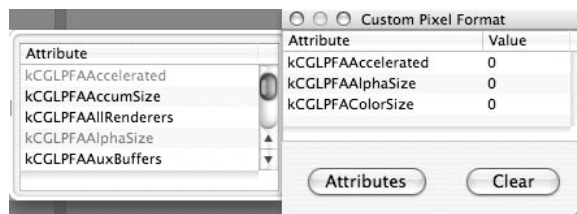
Unless you specify otherwise, OpenGL Profiler uses the native driver, the installed driver for your Mac's graphics card. By using the Choose driver pop-up, you can tell OpenGL Profiler to run your program on an emulated driver of another graphics card. The emulated drivers are just text files. They contain the OpenGL extensions the driver supports and the values of OpenGL state variables that depend on the OpenGL implementation. Examples of OpenGL state variables include the maximum size of a texture and the maximum number of instructions a shader can have. When you use an emulated driver, OpenGL profiler replaces your card's extensions and state variables with the emulated driver's extensions and state variables.

Using an emulated graphics card driver is not as cool as it sounds. To take advantage of a feature on another driver, your graphics card must support the feature. If your graphics card doesn't support a particular OpenGL extension, using a driver that supports the extension isn't going to allow your card to use that extension. Emulated drivers work better when your graphics card is better than the card you want to emulate. Using emulated drivers also helps test your card's driver on older versions of Mac OS X.

To examine a graphics card driver's capabilities, choose the card's driver from the Choose driver pop-up menu and click the View button. The emulation inspector window opens, which shows the OpenGL extensions the driver supports as well as the values of implementation-dependent OpenGL state variables.

## Setting Environment Variables

To add an environment variable, click the + button below the environment variable list. Enter the variable's name in the Name column and the variable's value in the Value column.



**Figure 13.2**

Custom pixel format window.

## Remote Profiling

The host is the computer that is going to run your OpenGL program. Initially the host is your computer. If your computer has a network connection to other Macs, you can change the host. Changing the host allows you to run your program on one computer and run OpenGL Profiler on the other. Running your program on one computer and OpenGL Profiler on the other helps tremendously if your program runs fullscreen. Profiling a fullscreen program on a standalone computer is difficult because you can't see the OpenGL Profiler windows when your program runs. Using two Macs lets you see the OpenGL Profiler windows without having to pause your fullscreen program.

If you can change hosts, there will be a Change Hosts button in the starting point dialog. Click the Change Hosts button. A list of available hosts will appear. Select a host from the list.

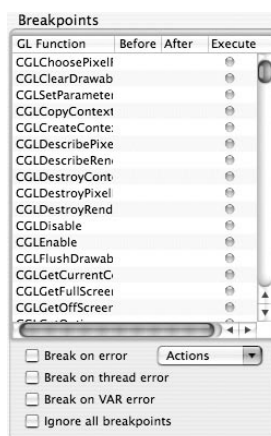
## Setting Breakpoints

To be able to perform certain tasks, such as viewing the contents of OpenGL buffers, you have to set breakpoints in OpenGL Profiler. Choose Views > Breakpoints to open the breakpoints window, which you can see in Figure 13.3. You can set a breakpoint at any OpenGL function call and at any Core OpenGL (CGL) function call. CGL is the way Mac OS X programs access OpenGL. If you use the Cocoa OpenGL classes or use AGL in Carbon, you're indirectly using CGL; AGL and the Cocoa classes sit on top of CGL.

On the left side of the window, you'll see an alphabetical list of function calls with three columns: Before, After, and Execute. All the functions initially have the Execute column selected, which means the functions execute normally. Disabling the Execute column for a function means OpenGL Profiler skips the function when it finds a call to that function in your program. Core OpenGL functions must execute so OpenGL Profiler prohibits you from disabling Execute for those functions.

You can set a breakpoint before an OpenGL function call, after, or both. Setting a breakpoint before and after a function lets you see the function's effects on OpenGL's buffers. Click the Before or After column to set a breakpoint on a function. A blue dot signifies that you set a breakpoint.

In addition to setting breakpoints for individual functions, you can tell OpenGL Profiler to pause execution when your program commits an OpenGL error. Below the function list is a series of checkboxes to break on various types of OpenGL errors. Next to the checkboxes is the Actions pop-up menu. Use the Actions pop-up menu to set and remove breakpoints for all functions.



**Figure 13.3**

Breakpoints window.

## Profiling Your Program

Before you start to profile, you must tell OpenGL Profiler what to record as your program runs. There are two checkboxes in the upper right corner of the main window. Selecting the Collect Trace checkbox tells OpenGL Profiler to record a list of all the OpenGL function calls your program makes as it runs.

Selecting the Include backtraces checkbox tells OpenGL Profiler to record the call stack. If you set breakpoints, you can examine the call stack.

Click the Launch button to launch your program. The program continues to run until it hits a breakpoint or you click the Suspend button. Click the Resume button to continue profiling. Clicking the Kill button quits the program you're profiling.

## Viewing the Profiling Data

The main OpenGL Profiler window doesn't provide much profiling data about your program. It tells you the current and peak frame rates. To see more data about your program's performance, you must use OpenGL Profiler's auxiliary windows, which you can open from the View menu.

### Trace Window

The trace window shows every OpenGL function call your program made. The window lists the function calls in the order your program made them, one function call per line. By examining the trace you can find unnecessary function calls, improving your program's performance. Looking at the function trace also helps you debug your program. You can learn your program is passing bad data to OpenGL functions and discover missing functions, functions your program should be calling but isn't calling.

The trace window has controls to show more data in the trace window. Selecting the Line #s checkbox adds a line number to each listing. Selecting the Context checkbox adds the OpenGL context to each listing. Selecting the Timing checkbox adds the amount of time your program spent in the function. The trace window reports the time in microseconds.

Clicking the Call Stack button opens the call stack drawer. Selecting a function from the trace window writes the function's call stack to the call stack drawer.

OpenGL Profiler provides two ways to save a trace to a text file. Clicking the Save As Text button saves the entire trace to a text file. Clicking the Filter button runs the trace file through a filter and saves the functions that pass through the filter to a text file. A *filter* is a shell script that weeds out functions you don't want to save. The trace window has a combo box that contains recently used filters and an Open button to let you choose a filter.

### Buffers Window

The buffers window lets you see the contents of the alpha, back, stencil, and depth buffers. To be able to view a particular buffer, you must be using it in your program. If you don't use the stencil buffer, there's not much point in being able to look at it with OpenGL Profiler. You can view the buffers only when OpenGL Profiler stops at a breakpoint so make sure you set some breakpoints.

## Resources Window

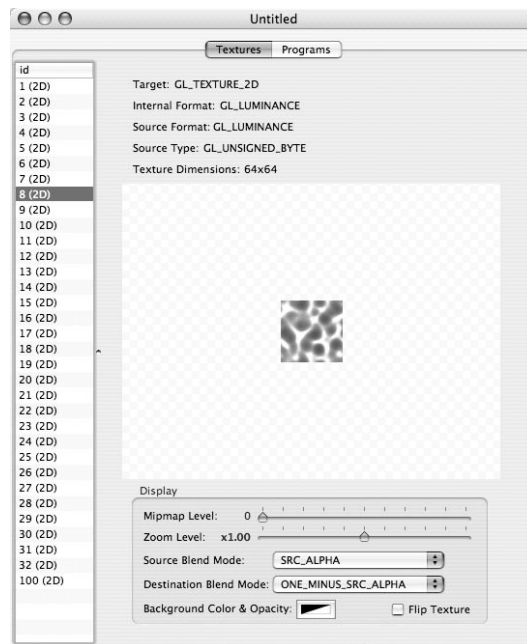
The resources window, shown in Figure 13.4, lets you examine the textures, vertex programs, and fragment programs your application uses. There are tabs in the resources window that let you toggle between viewing textures and viewing vertex and fragment programs. For vertex and fragment programs, the resources window contains a list of programs along the left side of the window. Select one from the list to view its contents.

You can do more with textures. Select a texture from the list running along the left side of the resources window. In the center of the resources window is the texture itself. Above the texture is information about the texture like its size, target (1, 2, or 3 dimensional texture), and format. Below the texture are options for displaying the texture in the resources window. These options include.

- Mipmap level.
- Zoom level, which lets you get a closer look at the texture.
- Source blend mode.
- Destination blend mode.
- Background color.
- Background opacity.

*Mipmaps* are smaller versions of a texture used to provide level of detail. When the object moves farther away from the camera, OpenGL uses a smaller version of the texture. In OpenGL you cut the size of the texture in half until you reach a 1-by-1 pixel mipmap. If the base texture is 64-by-64 pixels, the mipmaps would be 32-by-32, 16-by-16, 8-by-8, 4-by-4, 2-by-2, and 1-by-1. With OpenGL Profiler you can view these smaller textures.

The blending modes determine how the incoming fragment's color blends with the texture environment's color to create a final color for the pixel.



**Figure 13.4**

Resources window.

## Scripts Window

Use the scripts window, shown in Figure 13.5, to write scripts that execute when your program reaches a breakpoint. You can include any OpenGL function call in a script, but a script can contain only OpenGL function calls.

Click the + button to create a script. OpenGL Profiler adds it to the script list with the name `Unnamed`. Double-click the name to change the script's name. Use the text editor in the top half of the window to create the script. When you finish writing your script, click the Execute button to test the script for syntax errors.

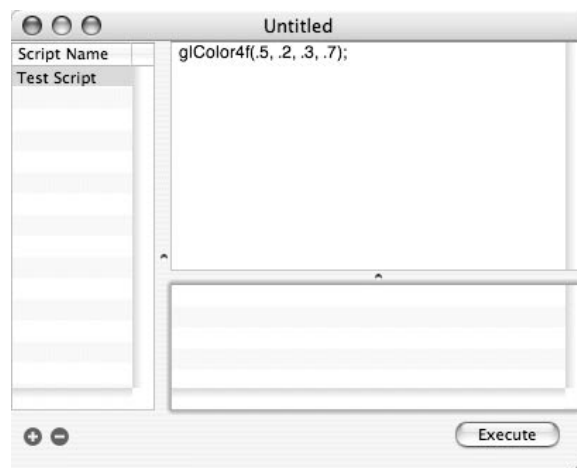
After you finish your script and fix any syntax errors, you must tell OpenGL Profiler to execute the script when your program hits a breakpoint.

- 1) Open the breakpoints window by choosing Views > Breakpoints.
- 2) Select a function from the list.
- 3) Choose Attach Script from the Actions pop-up menu.
- 4) A dialog box opens with a list of scripts to attach. Choose a script from the list.
- 5) Decide when you want the script to run: before or after reaching the function.
- 6) Click the Attach button.

To detach a script from a function, select the function from the function list and choose Remove Script from the Actions pop-up menu.

## Breakpoints Window

I covered the setting of breakpoints earlier, but there are two pieces of information that appear in the breakpoints window when your program reaches a breakpoint. The first piece of information is the call stack, the list of functions your program called leading up to the function where you set the breakpoint. The second piece is a snapshot of all the OpenGL state information when you hit the breakpoint. The call stack and OpenGL state information help tremendously during debugging. There are tabs to toggle between viewing the call stack and OpenGL state information, even though they don't appear in the breakpoints window shown in Figure 13.3.



**Figure 13.5**

Scripts window.

## Statistics Window

The statistics window, shown in Figure 13.6, shows the profiling statistics of the OpenGL functions your program calls. It tells you the total amount of time, measured in microseconds, your program spent in OpenGL function calls along with the percentage of time your program spent in OpenGL. For each OpenGL function your program calls, the statistics window tells you the following information:

- The number of times your program called the function.
- The total time, measured in microseconds, your program spent in the function.
- The average time, measured in microseconds, your program spent in the function.
- The percentage of the OpenGL time your program spent in the function.
- The percentage of time your program spent in the function.

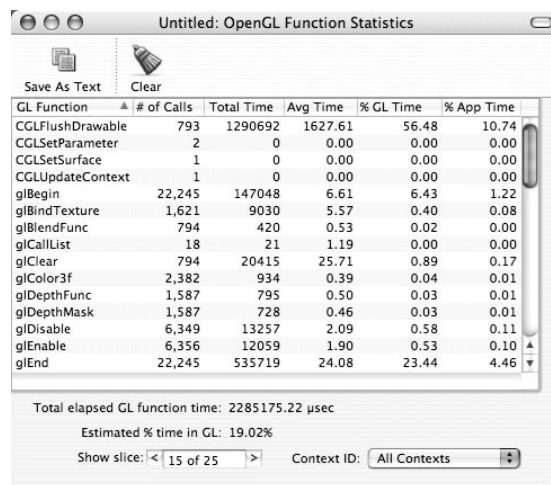
When OpenGL Profiler samples the OpenGL functions in your program, it breaks the samples into slices, 25 slices by default. If you sample your program for a short period of time, there may be fewer than 25 slices. The statistics for each slice build on the previous slice. If your program calls a function 20 times during the first slice and 15 times during the second slice, when you look at slice 2, it reports 35 function calls. Move slice by slice to see how your program performs over time.

## Pixel Format Window

The pixel format window displays the pixel format information for each OpenGL context your application uses. The information the pixel format window displays is the same information you can set when using custom pixel formats.

## Messages Window

The messages window displays the log messages OpenGL Profiler generates as your program runs. These messages can help when debugging your application.



The screenshot shows a window titled "Untitled: OpenGL Function Statistics". It contains a table with the following columns: GL Function, # of Calls, Total Time, Avg Time, % GL Time, and % App Time. The table lists various OpenGL functions and their corresponding statistics. Below the table, there is a summary section showing the total elapsed GL function time, estimated % time in GL, and controls for showing slices and context ID.

GL Function	# of Calls	Total Time	Avg Time	% GL Time	% App Time
CGLFlushDrawable	793	1290692	1627.61	56.48	10.74
CGLSetParameter	2	0	0.00	0.00	0.00
CGLSetSurface	1	0	0.00	0.00	0.00
CGLUpdateContext	1	0	0.00	0.00	0.00
glBegin	22,245	147048	6.61	6.43	1.22
glBindTexture	1,621	9030	5.57	0.40	0.08
glBlendFunc	794	420	0.53	0.02	0.00
glCallList	18	21	1.19	0.00	0.00
glClear	794	20415	25.71	0.89	0.17
glColor3f	2,382	934	0.39	0.04	0.01
glDepthFunc	1,587	795	0.50	0.03	0.01
glDepthMask	1,587	728	0.46	0.03	0.01
glDisable	6,349	13257	2.09	0.58	0.11
glEnable	6,356	12059	1.90	0.53	0.10
glEnd	22,245	535719	24.08	23.44	4.46

Total elapsed GL function time: 2285175.22  $\mu$ sec  
 Estimated % time in GL: 19.02%  
 Show slice: < 15 of 25 > Context ID: All Contexts

**Figure 13.6**

Statistics window.

## OpenGL Driver Monitor

OpenGL Driver Monitor collects statistics about your graphics card and its interaction with the CPU. It works at a lower level than OpenGL Profiler and works with the graphics card instead of a single application. Use OpenGL Driver Monitor to learn about things like how much video memory you're using, how much time the CPU spent waiting for the graphics card, the number of times the card swapped buffers, and the number of textures loaded on the card.

### Getting Started

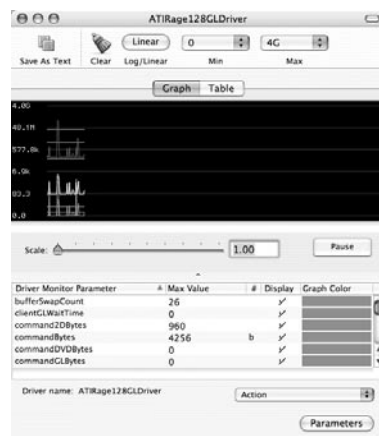
The first thing to do when you launch OpenGL Driver Monitor is display the monitor window. Choose **Monitors > Driver Monitors > DriverName** to open the monitor window, which you can see in Figure 13.7. Initially there's nothing to look at because you haven't given OpenGL Driver Monitor any parameters to monitor. Click the **Parameters** button to open the parameters drawer. Double-click a parameter to add it to the watch list, which contains the parameters whose statistics you want to collect. You can also select parameters from the drawer and drag them to the parameter watch list at the bottom of the monitor window.

Now that you've added parameters to watch, run your OpenGL program. Lines should start appearing on the graph, which means OpenGL Driver Monitor is collecting statistics. If no lines appear, make sure the button below the graph says **Pause** and not **Continue**. If it says **Continue**, click the button. OpenGL Driver Monitor's default sampling rate is one second, which means it plots a point on the graph every second. You can change the sampling rate in the preferences window.

### Customizing the Graph

Watching multiple parameters on the graph can be difficult because OpenGL Driver Monitor gives each parameter the same color: green. The parameter watch list is where you change a parameter's color on the graph. The list has five columns.

- Color.
- Display. If the Display column has a check mark, the parameter appears on the graph.
- Driver Monitor Parameter, the name of the parameter.
- Max Value, the highest value OpenGL Driver Monitor has sampled.
- #, the unit of measurement. Common units are **b** for bytes and **ns** for nanoseconds.



**Figure 13.7**

OpenGL Driver Monitor window.

Watch list

To change a parameter's color, select it from the watch list. Double-click the Graph Color column. A window opens for you to set the parameter's color. Give each parameter its own color to make the graph easier to read.

There are pop-up menus at the top of the window that let you set the minimum and maximum values the graph displays. The Linear/Log button controls the scale of the graph, the values running along the left side of the graph. The linear scale (the button says Log when showing the linear scale) has even scales. The logarithmic scale (the button says Linear) uses uneven scales to fit as much data in the graph as possible. The data determines the scales in the logarithmic scale while the minimum and maximum values determine the scales in the linear scale. Look at Figure 13.8 to see the difference between the two scales.

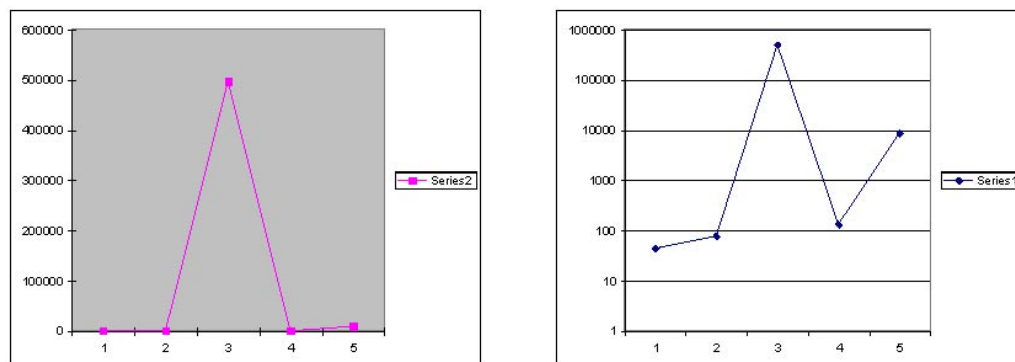
## Table View

If you don't want to fiddle with graph colors and just want to see the raw data, use the table view. Click the Table tab to switch from the graph to the table view. There's one column for each parameter in the watch list. After each sampling period, one second by default, OpenGL Driver Monitor adds a row to the table that contains the newly sampled data.

## Renderer Info

Choosing Monitors > Renderer Info opens the renderer window, which displays information about the OpenGL renderers on your Mac. Every Mac has at least two renderers: the renderer for the graphics card and the software renderer, which appears as **Generic** in the window. OpenGL Driver Monitor reports lots of information for a renderer, including the following:

- The amount of video memory.
- The amount of texture memory.
- The OpenGL extensions it supports.
- The available sizes of the depth and stencil buffers.
- The available color depths for the color and accumulation buffers.



**Figure 13.8**

Linear (left) and logarithmic scales for the values 44, 76, 497216, 131, and 8671.



## OpenGL Shader Builder

OpenGL Shader Builder lets you create, test, and debug shaders. Shaders are programs that are meant to run on the graphics card instead of the CPU. The shader replaces a portion of the fixed-function OpenGL pipeline, giving you more control and flexibility in drawing a scene.

There are two types of shaders: vertex programs and fragment programs. A *vertex program* gives you control over the transformation (converting a scene from world space to screen space) and lighting stage of the 3D graphics pipeline. Instead of feeding vertices to OpenGL and letting it take care of the transformation and lighting, each vertex goes through the vertex program. Tasks you can perform in a vertex program include:

- Vertex transformation, weighting, and blending.
- Normal transformation, rescaling, and normalization.
- Texture coordinate generation.
- Texture coordinate transformation.
- Lighting.
- Color material application.

A vertex program doesn't have to do all the tasks in the list above, but if you don't perform one of the tasks in the vertex program, OpenGL won't perform the task for you. If your vertex program generates texture coordinates, OpenGL won't transform the texture coordinates for you. You'll have to transform them yourself.

*Fragment programs*, sometimes referred to as pixel shaders, operate on fragments. You can think of a fragment as a pixel wannabe. The fragment contains data for a pixel, such as its primary color and position. OpenGL performs a series of tests on each fragment to determine whether or not the fragment becomes a pixel. Fragment programs generate a final color and a depth value for the fragment, from which OpenGL performs its tests. Fragment programs can perform the following tasks:

- Texture access.
- Texture application.
- Color sum, which creates a final color from the fragment's primary and secondary colors.
- Fog, blending a fog color with the fragment's color.
- Operations on interpolated values in the texture, color sum, and fog stages.

A fragment program doesn't have to do all the tasks in the list above, but OpenGL won't perform the task for you if the fragment program doesn't do it. If your fragment program performs fog operations, OpenGL won't perform color summing for you. You'll have to do any color summing yourself.

## Writing a Shader

OpenGL Shader Builder supports vertex programs written with the OpenGL extension `ARB_vertex_program` and fragment programs written with the extensions `ARB_fragment_program` and `ATI_text_fragment_shader`. These three extensions use a language that resembles assembly language. I recommend using `ARB_fragment_program` instead of `ATI_text_fragment_shader` for fragment programs. `ATI_text_fragment_shader` works only on Macs and only on Macs with ATI cards. `ARB_fragment_program` works on multiple operating systems with the graphics cards of multiple vendors.

The version of OpenGL Shader Builder that shipped with Mac OS X 10.4 added support for the OpenGL Shading Language (GLSL). GLSL resembles the C programming language and makes writing shaders easier. I'm going to focus on using the `ARB_fragment_program` and `ARB_vertex_program` extensions, even though I'm aware GLSL is the future of OpenGL shader programming. GLSL is too large for me to cover adequately in this chapter. For those of you interested in GLSL, you can download documentation about it from OpenGL's website. The documentation is excellent, which is another reason for me to focus on the ARB extensions.

OpenGL Shader Builder provides a reference for the statements in vertex and fragment programs so I won't rehash them here. I'm going to cover the areas of writing vertex and fragment programs the reference doesn't cover. The Xcode Tools ship with several shader examples to help you start writing your own vertex and fragment programs. You can find them at `/Developer/Examples/OpenGL/Shaders`.

## Starting and Ending a Shader

The first statement of a shader is a statement describing the type of program being written and its version. A vertex program written using `ARB_vertex_program` has the first statement.

```
!!ARBvp1.0
```

A fragment program written using `ARB_fragment_program` has the first statement.

```
!!ARBfp1.0
```

The `END` statement is the last statement in a shader.

```
END
```

## Placing Comments

The character `#` signals a comment. OpenGL treats everything after the `#` character in a line of code as a comment.

```
# This is a comment
```

Shaders written with `ARB_vertex_program` and `ARB_fragment_program` are difficult to read. You should write plenty of comments to make your shader code easier for other people to understand.

## Declaring Variables

Declaring variables is an important part of any program, including vertex and fragment programs. There are four kinds of variables you can declare in a vertex or fragment program.

- Attribute variables store the attributes for the vertex or fragment your OpenGL program passes to the shader.
- Program parameter variables store parameters your OpenGL program sends to the shader.
- Temporary variables store temporary results as the shader executes.
- Result variables store the final results of the shader.

## Attribute Variables

Use the `ATTRIB` statement to declare an attribute variable. For a vertex program the attribute variable's value must be one of the vertex attributes listed in Table 13.1.

```
ATTRIB vertexPosition = vertex.position;
```

**Table 13.1 Vertex Attributes**

Attribute	Components	Description
<code>vertex.position</code>	(x, y, z, w)	The vertex's position.
<code>vertex.weight[n]</code>	(w, w, w, w)	The vertex's weight, starting with weight n. If you don't supply weight n, the program uses weights 0-3.
<code>vertex.normal</code>	(x, y, z, 1)	The vertex's normal.
<code>vertex.color.primary</code>	(r, g, b, a)	The vertex's primary color.
<code>vertex.color.secondary</code>	(r, g, b, a)	The vertex's secondary color.
<code>vertex.fogcoord</code>	(f, 0, 0, 1)	The vertex's fog coordinate.
<code>vertex.texcoord[n]</code>	(s, t, r, q)	The vertex's texture coordinate for texture unit n. If you don't supply texture unit n, the program uses texture unit 0.
<code>vertex.matrixindex[n]</code>	(i, i, i, i)	The vertex's matrix indices, starting with index n. If you don't supply index n, the program uses indices 0-3.
<code>vertex.attrib[n]</code>	(x, y, z, w)	If you don't want to use the attribute names in this table, you can use the generic attributes listed in Table 13.2.

**Table 13.2 Generic Attribute Equivalents**

Vertex Attribute	Generic Equivalent
<code>vertex.position</code>	<code>vertex.attrib[0]</code>
<code>vertex.weight</code>	<code>vertex.attrib[1]</code>
<code>vertex.normal</code>	<code>vertex.attrib[2]</code>
<code>vertex.color.primary</code>	<code>vertex.attrib[3]</code>
<code>vertex.color.secondary</code>	<code>vertex.attrib[4]</code>
<code>vertex.fogcoord</code>	<code>vertex.attrib[5]</code>
<code>vertex.texcoord[n]</code>	<code>vertex.attrib[8 + n]</code>

For a fragment program the attribute variable's value must be one of the fragment attributes listed in Table 13.3.

```
ATTRIB fragmentFog = fragment.fogcoord;
```

Table 13.3 Fragment Attributes

Attribute	Components	Description
<code>fragment.color.primary</code>	(r, g, b, a)	The fragment's primary color.
<code>fragment.color.secondary</code>	(r, g, b, a)	The fragment's secondary color.
<code>fragment.texcoord[n]</code>	(s, t, r, q)	The fragment's texture coordinate for texture unit n. If you don't supply texture unit n, the program uses texture unit 0.
<code>fragment.fogcoord</code>	(f, 0, 0, 1)	The fragment's fog coordinate (distance).
<code>fragment.position</code>	(x, y, z, 1/w)	The fragment's window position.

### Program Parameter Variables

Use the `PARAM` statement to declare program parameter variables. Supply an environment variable or local variable listed in Table 13.4.

```
PARAM direction = program.env[1];
```

If you supply an array of variables, the parameter variable must be an array as well.

```
PARAM x[3] = program.local[2..4];
```

Table 13.4 Program Parameters

Parameter	Components	Description
<code>program.env[a]</code>	(x, y, z, w)	Program environment parameter a.
<code>program.local[a]</code>	(x, y, z, w)	Program local parameter a.
<code>program.env[a..b]</code>	(x, y, z, w)	Program environment parameters a through b.
<code>program.local[a..b]</code>	(x, y, z, w)	Program local parameters a through b.

Where do you come up with the indices for the environment and local variables? You control them with the `glProgramEnvParameter()` and `glProgramLocalParameter()` series of functions. The sample below sets the environment parameter index to 3 for a vertex program.

```
GLfloat param[4] = { .21f, .45f, .05f, .55f };
glProgramEnvParameter4fvARB(GL_VERTEX_PROGRAM_ARB, 3, param);
```

A name like `program.env[0]` doesn't tell you what you're passing to the shader. To provide a more descriptive name, pass OpenGL state variables as parameters. You can pass the following types of state variables as parameters to vertex and fragment programs:

- Material
- Lighting
- Texture coordinates
- Texture environments
- Fog
- Clipping planes

- Points
- Depth
- Matrices

## Material Variables

Use material state variables to pass the material settings to your vertex or fragment program. Table 13.5 lists the available material state variables.

```
PARAM shininess = state.material.front.shininess;
```

Call the `glMaterial()` series of functions in your OpenGL program to set the variables in Table 13.5.

```
glMaterialf(GL_FRONT, GL_SHININESS, .43f);
```

**Table 13.5 Material Variables**

Variable	Components	Description
<code>state.material.front.ambient</code>	(r, g, b, a)	Front ambient material color.
<code>state.material.front.diffuse</code>	(r, g, b, a)	Front diffuse material color.
<code>state.material.front.specular</code>	(r, g, b, a)	Front specular material color.
<code>state.material.front.emission</code>	(r, g, b, a)	Front emissive material color.
<code>state.material.front.shininess</code>	(s, 0, 0, 1)	Front material shininess.
<code>state.material.back.ambient</code>	(r, g, b, a)	Back ambient material color.
<code>state.material.back.diffuse</code>	(r, g, b, a)	Back diffuse material color.
<code>state.material.back.specular</code>	(r, g, b, a)	Back specular material color.
<code>state.material.back.emission</code>	(r, g, b, a)	Back emissive material color.
<code>state.material.back.shininess</code>	(s, 0, 0, 1)	Back material shininess.

## Light Variables

Use light state variables to pass the light settings to your vertex or fragment program. Table 13.6 lists the available light state variables.

```
PARAM position0 = state.light[0].position;
```

Call the `glLight()` or `glLightModel()` series of functions in your OpenGL program to set the variables in Table 13.6.

```
GLfloat lightPosition0[] = { .25f, .61f, .08f };
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
```

Table 13.6 Light Variables

Variable	Components	Description
<code>state.light[n].ambient</code>	(r, g, b, a)	Light n's ambient color.
<code>state.light[n].diffuse</code>	(r, g, b, a)	Light n's diffuse color.
<code>state.light[n].specular</code>	(r, g, b, a)	Light n's specular color.
<code>state.light[n].position</code>	(x, y, z, w)	Light n's position.
<code>state.light[n].attenuation</code>	(a, b, c, e)	Light n's attenuation constant.
<code>state.light[n].spot.direction</code>	(x, y, z, c)	Light n's spotlight direction and cutoff angle cosine.
<code>state.light[n].half</code>	(x, y, z, l)	Light n's infinite half angle.
<code>state.lightmodel.ambient</code>	(r, g, b, a)	Light model's ambient color.
<code>state.lightmodel.front.scenecolor</code>	(r, g, b, a)	Light model's front scene color.
<code>state.lightmodel.back.scenecolor</code>	(r, g, b, a)	Light model's back scene color.
<code>state.lightprod[n].front.ambient</code>	(r, g, b, a)	Light n's front material ambient color product.
<code>state.lightprod[n].front.diffuse</code>	(r, g, b, a)	Light n's front material diffuse color product.
<code>state.lightprod[n].front.specular</code>	(r, g, b, a)	Light n's front material specular color product.
<code>state.lightprod[n].back.ambient</code>	(r, g, b, a)	Light n's back material specular color product.
<code>state.lightprod[n].back.diffuse</code>	(r, g, b, a)	Light n's back material diffuse color product.
<code>state.lightprod[n].back.specular</code>	(r, g, b, a)	Light n's back material specular color product.

### Texture Coordinate Generation Variables

Use the texture coordinate generation state variables to pass the texture unit's linear plane coefficients to your shader. Only vertex programs use texture coordinate generation state variables. Table 13.7 lists the available texture coordinate generation state variables.

```
PARAM eyeT = state.texgen[0].eye.t;
```

Call the `glTexGen()` series of functions in your OpenGL program to set the variables in Table 13.7.

```
GLfloat eyeTParameter[] = { .11f, .22f, .33f, .05f };
glTexGenfv(GL_T, GL_EYE_PLANE, eyeTParameter);
```

**Table 13.7 Texture Coordinate Generation Variables**

Variable	Components	Description
<code>state.texgen[n].eye.s</code>	(a, b, c, d)	The <i>s</i> coordinate of the eye linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].eye.t</code>	(a, b, c, d)	The <i>t</i> coordinate of the eye linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].eye.r</code>	(a, b, c, d)	The <i>r</i> coordinate of the eye linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].eye.q</code>	(a, b, c, d)	The <i>q</i> coordinate of the eye linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].object.s</code>	(a, b, c, d)	The <i>s</i> coordinate of the object linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].object.t</code>	(a, b, c, d)	The <i>t</i> coordinate of the object linear plane coefficients for texture unit <i>n</i> .
<code>state.texgen[n].object.r</code>	(a, b, c, d)	The <i>r</i> coordinate of the object linear plane coefficients.
<code>state.texgen[n].object.q</code>	(a, b, c, d)	The <i>q</i> coordinate of the object linear plane coefficients.

### Texture Environment Variables

Use the texture environment state variables to pass the texture environment's color to your shader. Only fragment programs use texture environment state variables. Table 13.8 lists the available texture environment state variables.

```
PARAM textureColor = state.texenv[0].color;
```

Call the `glTexEnv()` series of functions in your OpenGL program to set the texture environment's color.

```
GLfloat color[] = {.85f, .1f, .05f, .03f};
glTexEnvfv (GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, color);
```

**Table 13.8 Texture Environment Variables**

Variable	Components	Description
<code>state.texenv[n].color</code>	(r, g, b, a)	The color for texture environment <i>n</i> .

### Fog Variables

Use the fog state variables to pass attributes of your program's fog to your vertex or fragment program. Table 13.9 lists the available fog state variables.

```
PARAM fogColor = state.fog.color;
```

Call the `glFog()` series of functions in your OpenGL program to set the variables in Table 13.9.

```
GLfloat color[] = {.75f, .12f, .25f, .03f};
glFogfv(GL_FOG_COLOR, color);
```

**Table 13.9 Fog Variables**

Variable	Components	Description
state.fog.color	(r, g, b, a)	The fog color.
state.fog.params	(d, s, e, r)	The fog's density, start, end, and the value (1 / (end - start)).

### Clipping Plane Variables

Use the clipping plane state variables to pass properties of the clipping plane to your vertex or fragment program. Table 13.10 lists the available clipping plane state variables.

```
PARAM clipPlaneCoeff = state.clip[1].plane;
```

Call the function `glClipPlane()` in your OpenGL program to set the clipping plane properties.

```
glDouble equation[] = {.35, .46, .57, .68 };
glClipPlane(GL_CLIP_PLANE1, equation);
```

**Table 13.10 Clipping Plane Variables**

Variable	Components	Description
state.clip[n].plane	(a, b, c, d)	The coefficient for clip plane n.

### Point Variables

OpenGL version 1.4 introduced point parameters, which let programmers specify attributes of a point, like its minimum and maximum sizes. Use point state variables to pass point parameters to your shader. Only vertex programs use point state variables. Table 13.11 lists the available point state variables.

```
PARAM attenuation = state.point.attenuation;
```

Call the `glPointParameter()` series of functions in your OpenGL program to set the variables in Table 13.11.

```
glFloat attenuation[] = {.5f, .4f, .75f };
glPointParameterfv(GL_POINT_DISTANCE_ATTENUATION, attenuation);
```

**Table 13.11 Point Variables**

Variable	Components	Description
state.point.size	(s, n, x, f)	The point's size, minimum size, maximum size, and fade threshold.
state.point.attenuation	(a, b, c, l)	The point size attenuation constants.



## Depth Variables

Use the depth state variables to pass the range of allowable depth values to your shader. Only fragment programs use depth state variables. Table 13.12 lists the available depth property variables.

```
PARAM range = state.depth.range;
```

Call the function `glDepthRange()` in your OpenGL program to set the depth range.

```
glDepthRange(.15, .85);
```

**Table 13.12 Depth Variables**

Variable	Components	Description
<code>state.depth.range</code>	(n, f, d, l)	The depth range's near, far, and (far - near) values.

## Matrix Variables

Use matrix variables to pass OpenGL's matrices to your vertex or fragment program. Table 13.13 lists the available matrix state variables.

```
PARAM projectionMatrix = state.matrix.projection;
```

Call the function `glMatrixMode()` in your OpenGL program to set the variable in Table 13.13.

```
glMatrixMode(GL_PROJECTION);
```

**Table 13.13 Matrix Variables**

Variable	Description
<code>state.matrix.modelview[n]</code>	Modelview matrix n.
<code>state.matrix.projection</code>	Projection matrix.
<code>state.matrix.mvp</code>	Modelview-projection matrix .
<code>state.matrix.texture[n]</code>	Texture matrix n.
<code>state.matrix.palette[n]</code>	Modelview palette matrix n.
<code>state.matrix.program[n]</code>	Program matrix n.

### Temporary Variables

Temporary variables come in handy when you must perform a series of calculations to get the final result. Store the intermediate results in temporary variables and your program becomes easier to understand. Use the `TEMP` statement to declare a temporary variable.

```
TEMP direction;
```

### Result Variables

Use the `OUTPUT` statement to declare a result variable. The result variable's value must be one of the attributes listed in Table 13.14 if you're writing a vertex program.

```
OUTPUT vertexPosition = result.position;
```

**Table 13.14 Vertex Program Result Attributes**

Attribute	Components	Description
<code>result.position</code>	(x, y, z, w)	The vertex's position in clip coordinates.
<code>result.color.front.primary</code>	(r, g, b, a)	The vertex's front facing primary color.
<code>result.color.front.secondary</code>	(r, g, b, a)	The vertex's front facing secondary color.
<code>result.color.back.primary</code>	(r, g, b, a)	The vertex's back facing primary color.
<code>result.color.back.secondary</code>	(r, g, b, a)	The vertex's back facing secondary color.
<code>result.fogcoord</code>	(f, *, *, *)	The vertex's fog coordinate.
<code>result.pointsize</code>	(s, *, *, *)	The vertex's point size.
<code>result.texcoord[n]</code>	(s, t, r, q)	The vertex's texture coordinate for texture unit n. If you don't supply texture unit n, the program uses texture unit 0.

The result variable's value must be one of the attributes listed in Table 13.15 if you're writing a fragment program.

```
OUTPUT fragmentDepth = result.depth;
```

**Table 13.15 Fragment Program Result Attributes**

Attribute	Components	Description
<code>result.color</code>	(r, g, b, a)	The fragment's color.
<code>result.depth</code>	(*, *, d, *)	The fragment's depth coordinate.

## Using OpenGL Shader Builder

When you launch OpenGL Shader Builder, two windows open: one large window and one small window. The small window is the OpenGL rendering window, which lets you see the effects of the shader on a sphere, teapot, or geometric plane. The rendering window doesn't show anything until you write the shader. The point of having the rendering window is to test your shaders without having to write code to load a model.

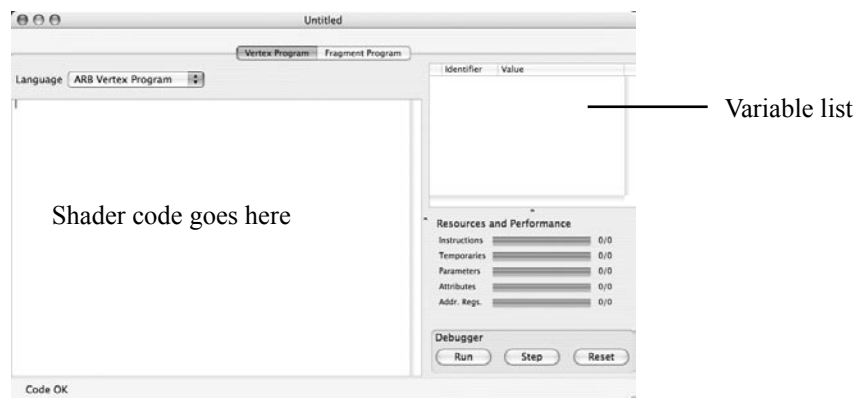
The large window is the shader window, shown in Figure 13.9. An OpenGL Shader Builder file can contain one vertex program and one fragment program. Some fragment programs require a vertex program in order to run so including a vertex and a fragment program in one OpenGL Shader Builder file makes sense. Use the Vertex Program and Fragment Program tabs to switch to the corresponding program. Use the Language pop-up menu to choose the language you want to use to write your shaders.

Use the large area taking up the left half of the shader window to edit your programs. OpenGL Shader Builder provides constant feedback, letting you know when you make a syntax error. The constant feedback can be annoying because OpenGL Shader Builder reports errors before you can finish a line of code. Finish typing your shader's code before looking for syntax errors.

Choosing Window > Instruction Reference opens the instruction reference window, which provides information about the instructions you can use in your shaders. The top half of the instruction reference window lists the available instructions. Vertex and fragment programs have different sets of instructions. The set of instructions that appears in the instruction reference window depends on the tab you select in the shader window. Selecting an instruction from the instruction list fills the bottom half of the window with information about the instruction. Double-clicking an instruction inserts the instruction into your program. Replace the placeholder operands with your shader's variables.

The upper right corner of the shader window contains the variable list, the variables your shader uses. OpenGL Shader Builder inserts the variables in the variable list as you add them to your shader. For each variable in your shader, the variable list shows the following information:

- The type of variable: attribute, program parameter, temporary, or result.
- The variable's name.
- The variable's value.



**Figure 13.9**

OpenGL Shader Builder window.

## Giving Your Variables Initial Values

When you declare a variable in your shader program, OpenGL Shader adds the variable to the variable list. Each variable in a shader is a vector with four components: *x*, *y*, *z*, and *w*. Initially each variable has zero as the value of its four components. If you want to change the initial value of a variable, select the variable from the variable list and choose Window > Symbol Editor.

You can directly set the components from the symbol editor window by typing in values or using the sliders. If you click the Color tab you can set the variable's components as a color with *x* representing the red component of the color, *y* the green component, *z* the blue component, and *w* the alpha component.

## Applying a Texture Map

The texture units window, which you can open by choosing Window > Texture Units, lets you load a texture to apply to the sphere, teapot, and geometric plane in the OpenGL rendering window. Click the Load button to load a texture map. Make sure you select the Enabled checkbox. If the checkbox is not selected, OpenGL Shader Builder won't apply your texture to the sphere, teapot, and plane.

## GLSL Log Window

If you're writing a shader using the OpenGL Shading Language, the GLSL log window provides information to help you. Choose Window > GLSL Log Window to open the log window. The log window has three tabs. The Errors tab lists the syntax errors in your shader. The Intermediate tab lists all the intermediate steps your GLSL code takes. Each line of GLSL code performs multiple actions behind the scenes. The Intermediate tab lets you see the actions.

The Assembly tab shows the assembly language the GLSL compiler generates. The assembly language does not run on any graphics cards. Do not confuse the assembly language the GLSL compiler generates with the `ARB_vertex_program` and `ARB_fragment_program` extensions. The GLSL compiler does not generate `ARB_vertex_program` and `ARB_fragment_program` shaders.

## Debugging Shaders

Debugging the vertex and fragment programs you write is easy in OpenGL Shader Builder because the assembly language shaders execute sequentially. Click the Run button to start debugging. Click the Step button to step through a line of code. When a line of code changes the value of one of your shader's variables, the change appears in the variable list. Click the Reset button when you're finished debugging to go back to editing your program.

## Moving Your Shader to Your OpenGL Program

After you finish writing your shaders, you must add them to an OpenGL application. If you don't have an OpenGL application, OpenGL Shader Builder can create one for you. Choose File > Export As Xcode Project. OpenGL Shader Builder creates an application project using the GLUT framework.

Exporting your shader as an Xcode project helps when you're starting out, but eventually you're going to want to add shaders to an existing OpenGL program. The hardest part of adding shaders to an OpenGL application is reading the shaders from disk. The code to read the shader files depends on the programming language and frameworks you're using, but you'll make your life easier if you add the shader files to your Xcode application project. Xcode can't compile shader files, but adding the shader files to your project ensures the shader files appear in your application bundle. Various Apple frameworks, including Cocoa and Core Foundation, have functions to read files from application bundles.

After reading your shaders from disk, you must take the following steps to use the shaders in your OpenGL program:

- 1) Create program objects for your vertex and fragment programs.
- 2) Choose the active vertex and fragment program objects.
- 3) Compile the shaders.
- 4) Insert your shaders into the OpenGL pipeline.

To call vertex and fragment programs from your OpenGL application, you must create program objects to store your vertex and fragment programs. Call the function `glGenProgramsARB()` to create a program object.

```
glGenProgramsARB(numberOfShaders, shaderList);
```

You can create as many program objects as you want, with each program object storing a vertex or fragment program. You must select one of the program objects to be the active one, the one your OpenGL application uses, by calling the function `glBindProgramARB()`. There can be one active vertex program and one active fragment program at one time.

```
glBindProgramARB(GL_VERTEX_PROGRAM_ARB, shaderID);
```

Call the function `glProgramStringARB()` to compile a vertex or fragment program. OpenGL adds the compiled code to the active program object.

```
glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB,
    GL_PROGRAM_FORMAT_ASCII_ARB,
    strlen(shader), shader);
```

The variable `shader` is a string that contains the vertex or fragment program you read from disk.

Call the function `glEnable()` to turn on a vertex or fragment program. OpenGL inserts the active program object's shader into the OpenGL pipeline, replacing a portion of the fixed-function pipeline. Call `glDisable()` to remove a vertex or fragment program from the OpenGL pipeline and restore the fixed-function pipeline.

```
glEnable(GL_FRAGMENT_PROGRAM_ARB);
glEnable(GL_VERTEX_PROGRAM_ARB);
```

# Putting the spotlight on Mac OS X development

Whether you're a newbie or a veteran programmer, iDevGames and iDevApps provide you with up-to-date guidance on development topics, tools, programming languages, APIs, and more.

- Community forum
- Latest developer news
- Articles and product reviews
- Resources and source code
- Programming contests

To start developing Mac OS X applications and games today, visit [www.idevgames.com](http://www.idevgames.com) & [www.idevapps.com](http://www.idevapps.com) !

Helping you create great Mac software, one question at a time, iDevGames & iDevApps.

Visit the Xcode Tools Sensei homepage  
for the latest news and updates.

<http://www.meandmark.com/xcodebook.html>

Email your questions and comments.

[xcodebook@meandmark.com](mailto:xcodebook@meandmark.com)