

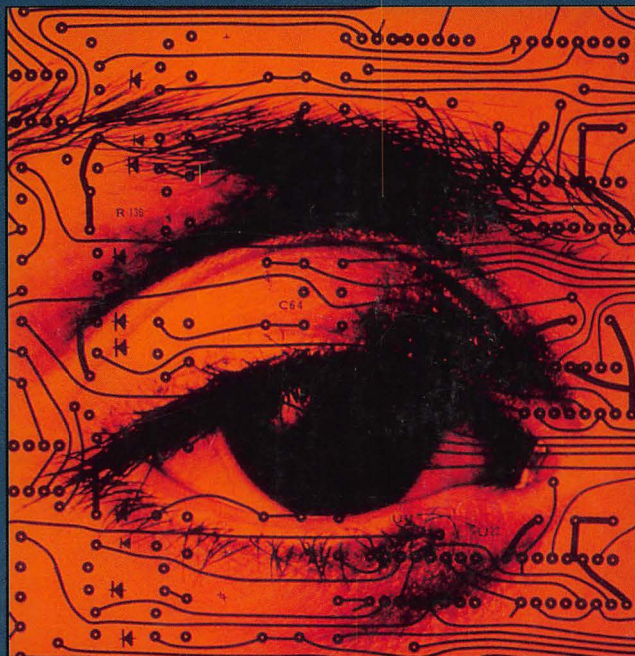
New Technology Building Blocks

Programming The PowerPC

Programming Native Applications for the New Power Macintosh



- Learn how to exploit the new RISC processors
- Develop native applications for the new Power Macs
- Learn to port existing programs from 680x0 to native PowerPC
- Examples in MetroWerks CodeWarrior and Symantec C++

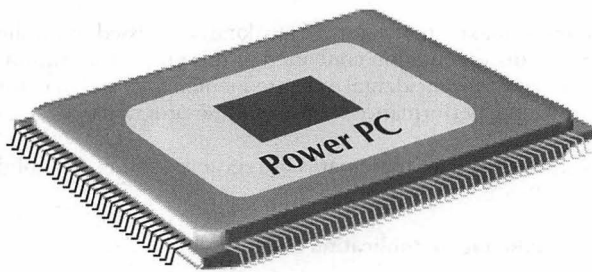


Series Editor: Tony Meadow
Bear River Associates



D a n P a r k s S y d o w

PROGRAMMING THE POWERPC



DAN PARKS SYDOW

M&T
 BOOKS

**M&T Books**

A Division of MIS:Press, Inc.

A Subsidiary of Henry Holt and Company, Inc.

115 West 18th Street

New York, New York 10011

© 1994 by M&T Books

Printed in the United States of America

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

Limits of Liability and Disclaimer of Warranty

The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All products, names and services are trademarks or registered trademarks of their respective companies.

Library of Congress Cataloging-in-Publication Data

Sydow, Dan P.

Programming the Power PC : programming native applications for the new power Macintosh / Dan Parks Sydow.

p. cm.

Johnson's name appears first on the earlier edition.

Includes index.

ISBN 1-55851-400-7 : \$34.95

1. Macintosh (Computer) --Programming. 2. PowerPC microprocessors--Programming. I. Title.

QA76.8.M3S965 1994

005.265--dc20

94-37488

CIP

97 96 95 94 4 3 2 1

Development Editor: Michael Sprague

Production Editor: Patricia Wallenburg

Copy Editors: Greg Robertson

Technical Editor: Peter Ferranti

DEDICATION

To my wife, Nadine...

Dan

ACKNOWLEDGMENTS

Anthony Meadow, Bear River Associates, for being the only person I know who can completely and accurately outline his thoughts for an entire book in one five minute conversation.

Michael Sprague, Development Editor, M&T Books, for his patience, sense of humor, patience, input, patience, encouragement, and, least I forget, for his patience.

Patty Wallenburg, Production Editor, M&T Books, for a page layout effort that resulted in such a polished looking book.

Peter Ferrante, Apple Computer, for another suggestion-filled technical book and software review.

Steve Devino, Teradyne, Inc., for additional software reviewing.

Dave Hirsh, for granting permission to include his data fork erasing utility program, DFerase, with this package.

Carole McClendon, Waterside Productions, for making this book happen.

William Barnekow, Professor, Milwaukee School of Engineering, for sparking my interest in computers and computer architecture.

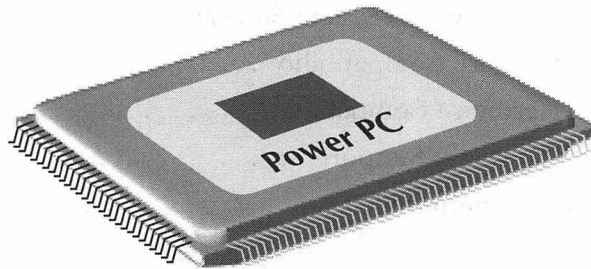


TABLE OF CONTENTS

Introduction	xvii
What's on the Disk	xix
What You Need	xix
Why This Book is for You	xx

CHAPTER 1

The PowerPC and the Power Macs	1
The Need for a New Chip	2
CISC and RISC	2
RISC Leads to More Than Just Speed	4
The Power Macintosh Line	5
Features of the New Macs	5
The Customer Base	8
Is It Still a Mac?	9

Programming the PowerPC

The PowerPC System Software	9
Software Compatibility	10
Hardware Compatibility	10
Developer Support	10
Chapter Summary	11

CHAPTER 2

CISC and RISC Technologies 13

CISC and the 680x0 Series	14
Why CISC?	14
Instruction Execution on a 680x0	14
The Timing of Instructions on a 680x0	17
CISC—Fast, But Not Fast Enough	20
RISC and the Power Mac Series	20
Why RISC?	20
Instruction Execution on a Power Mac	21
The Timing of Instructions on a Power Mac	25
Chapter Summary	27

CHAPTER 3

PowerPC Architecture 29

Branch Processing Unit	30
Instruction Fetching	30
Instruction Fetching and the Branch Unit	32
Superscaling	36
The Superscalar Design	37
Branch Processing Unit	38

Table of Contents

Integer Unit	38
Floating-Point Unit	38
Cache	39
Data Cache	39
Instruction Cache	41
Chapter Summary	43

CHAPTER 4

PowerPC System Software: The Emulator and Mixed Mode **45**

The PowerPC System Software	46
Ported System Software Routines	46
The New System Software	49
PowerPC Execution of System Software Routines ..	50
The 68LC040 Emulator	55
The Mixed Mode Manager	57
Instruction Set Architecture	58
Cross-Mode Calls	58
680x0 to PowerPC Cross-Mode Calls	59
PowerPC to 680x0 Cross-Mode Calls	60
The Programmer's Role in Mode-Switching	61
Chapter Summary	63

CHAPTER 5

PowerPC System Software Code Fragments **65**

The PowerPC Runtime Environment	66
What the Runtime Environment Is	66

Programming the PowerPC

A New Runtime Environment— And Why It Was Needed	66
Import Libraries	67
Linked Libraries and Import Libraries	68
Advantages of Import Libraries	69
Code Fragments	73
About Code Fragments	73
The Code Fragment Manager	74
Transition Vectors	78
The Table of Contents	79
Chapter Summary	84

CHAPTER 6

PowerPC Compilers **87**

The Metrowerks CodeWarrior Compilers	88
What Metrowerks Consists Of	88
Creating a CodeWarrior Project	90
Adding to the Project	92
The Prefix File	96
Creating the Resource File	98
The MWdemoPPC Source Code	100
Creating the PowerPC Application	103
Symantec's Cross-Development Kit (CDK)	105
What the CDK Consists Of	105
Installing AppleScript	106
Using AppleScript to Update ANSI Libraries	107
Creating a Folder to Hold Your Power Mac Project	109
Creating the Resource File	112

Table of Contents

Opening the CDK Project	115
Required Resources	119
The CDKdemoPPC Source Code	121
Creating the PowerPC Application	126
Chapter Summary	131

CHAPTER 7

Universal Procedure Pointers 133

Universal Procedure Pointer Theory	134
Procedure Pointers and the 680x0 Processor	134
Universal Procedure Pointers and the PowerPC ...	136
Using UniversalProcPtrs	140
Using a UniversalProcPtr in a Call to ModalDialog()	140
How the Compiler Chooses Between ProcPtr and UniversalProcPtr	144
Using UniversalProcPtrs In Other Toolbox Calls ..	146
UniversalProcPtr Example Programs	152
ModalDialog() and UPPs	152
Another Example of User Items and UPPs	157
Chapter Summary	164

CHAPTER 8

Fat Binary Applications 167

Fat Application Theory	168
Applications and 680x0/PowerPC Compatibility ..	168
Structure of a 680x0 Application	171

Programming the PowerPC

Structure of a PowerPC Application	173
Structure of a Fat Application	173
Using CodeWarrior to Create Fat Apps	176
Creating the PowerPC Version	176
Creating the 680x0 Version	179
Creating the Fat Binary	181
Using Symantec's CDK to Create Fat Apps	184
Creating the PowerPC Version	185
Creating the 680x0 Version	187
Creating the Fat Binary	190
Gracefully Exiting a PowerPC-only App	193
PowerPC-only Applications and User-Friendliness ..	194
The 680x0 Resource File	195
The 680x0 Source Code	196
Copying the Resources to the PowerPC-only App ..	197
Stripping Fat Applications	200
Converting a Fat Binary to a PowerPC Application ..	200
Converting a Fat Binary to a 680x0 Application ...	202
Chapter Summary	208

CHAPTER 9

The PowerPC Numerics Environment 209

Switching from SANE to PowerPC Numerics	210
PowerPC Numerics Data Formats	211
The Single Format	212
The Double Format	212
The Double-Double Format	212
Numeric Data Format Summary	212

Table of Contents

Numerics Libraries and the PowerPC	213
Numerics Porting Considerations	216
The extended and double_t Data Types	216
Eliminate the comp Data Type	218
Be Aware of How Expressions Are Evaluated	218
Chapter Summary	223
Chapter 10	225
Porting Code to Native PowerPC	225
Porting Preparation	226
Use the Universal Header Files	226
Change Assembly Code to C Code	229
ANSI C and the PowerPC	230
Change int Variables to Other Integral Types	230
Use ANSI Function Declarations	233
Use Function Prototypes	236
Using a Single Source File For	
Both 68K and PowerPC Development	237
Using Conditional Compilation Directives	238
QuickDraw Globals and Conditional	
Compilation Directives	239
How the Compiler Knows If powerc Is Defined	242
PowerPC Compatibility	248
Keep Code 32-bit Clean	248
Use Access Functions for Low-Memory Globals	249
Use Universal Procedure	
Pointers in Place of ProcPtrs	254
Data Alignment	255
The 680x0 Alignment Convention	256
The PowerPC Alignment Convention	258

Programming the PowerPC

Potential Data Alignment Problems	260
The Data Alignment Solution	261
Testing Data Alignment	263
Avoiding an Alignment Switch	267
Chapter Summary	271

CHAPTER 11

Import Libraries 273

Code Fragment Basics	274
All Code is a Fragment	274
Fragment Code and Containers	275
Import Library Basics	277
Imported and Exported Symbols	278
Import Library Special Routines	279
Import Library Code	282
Defining One of the Special Routines	282
A Second Initialization Routine Example	284
Import Library Advantages	286
Loading and Executing Import Library Code	287
Creating an FSSpec For an Import Library	288
Loading a Library	288
Unloading a Library	291
Creating a Library With CodeWarrior	291
The Import Library Resources	292
The Import Library Project	294
The Import Library Source Code	297
Creating a Test Application With CodeWarrior	300
The Application Resources	301

Table of Contents

The Application Project	302
The Application Source Code	304
Executing the Application and the Library	309
Loading a Library on Demand	312
The Test Application's Resources	313
The Argument for Import Libraries	315
The Test Application's Code	317
Sharing Import Libraries	324
Sharing the CompanyInfo Library Between Applications	324
Creating a 'shlb' Library	325
Chapter Summary	329

CHAPTER 12

More Import Libraries **331**

Adding Icons to Applications and Libraries	332
Adding an Icon to the Application	332
Adding an Icon to the Library	335
A Second Library Example	339
Opening a PICT File	340
The Initialization Routine	342
The Termination Routine	345
The Main Routine	346
Using CodeWarrior to Build the Library	347
Modifying the TestApp2 Application	349
Changes to the TestApp2 Resources	350
Changes to the TestApp2 Code	352
A Last Word on the Main Routine	358

Programming the PowerPC

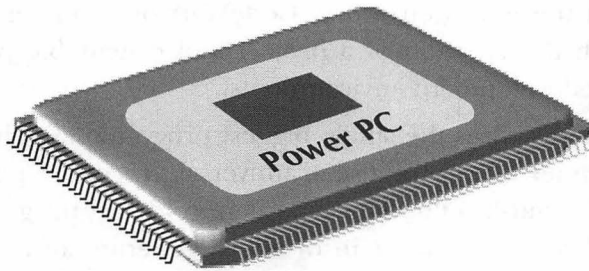
Testing the PICTchooser Library	360
Apple Events	362
Introduction to Apple Events	362
Responding to a Quit Application Apple Event	363
Adding Apple Events to an Application	367
Modifying the Main Event Loop	368
Installing the Event Handlers	369
Defining the Event Handlers	370
Defining the Open Document Event Handler	371
Testing Apple Events	375
Chapter Summary	376

CHAPTER 13

Optimizing PowerPC Code 379

Improving the Timing of WaitNextEvent()	380
Using WaitNextEvent()	
Outside the Main Event Loop	380
Verifying the WaitNextEvent()	
Is Time Consuming	382
A First Solution—Fewer Calls to WaitNextEvent() ..	385
A Second Solution—	
Timing the Calls to WaitNextEvent()	386
Miscellaneous Performance Enhancements	389
Align Data Structures	389
Move Floating-Point Parameters	
to the End of the List	389
Chapter Summary	390

Index 393



INTRODUCTION

Chapter 1 is an introduction to the new line of Macintosh computers—the Power Macintoshes. Here you'll see why the timing is right for Apple to switch to a new microprocessor.

Chapters 2 and 3 discuss the architecture of the new PowerPC microprocessor. First, the differences between the microprocessor design strategies of the 680x0 chips and the new PowerPC chips are described. Then, the architectural details that make the PowerPC such a powerful chip are discussed.

Chapters 4 and 5 cover the new additions to the Macintosh system software. Chapter 4 covers the 68LC040 Emulator. This built-in software is what allows a Power Mac to run both old 680x0 applications and new PowerPC programs. This chapter also covers the new Mixed Mode Manager—the part of the Toolbox that coordinates the activity of the 68LC040 Emulator. Chapter 5 covers the other manager new to the Power Macs—the Code Fragment Manager. Any executable code is now known as a fragment, and is handled by the Code Fragment Manager.

Chapter 6 describes the new PowerPC compilers that are on the market. Here you'll see a comparison of Symantec's Cross-Development Kit,

Programming the PowerPC

or CDK, and the new Metrowerks CodeWarrior compiler. This chapter steps through the creation of a project and executable program using both of these development environments.

Chapter 7 discusses the single biggest programming change a Power Mac programmer faces—the use of universal procedure pointers. Use of the new UniversalProcPtr data type is one of the programming techniques you'll need to master in order to generate an application that runs on both a 680x0-based Macintosh and a Power Mac.

Chapter 8 discusses fat binary applications—programs that run on both old and new Macs. You'll see how this type of application actually consists of two complete versions of the same program—one compiled with a 680x0 compiler and the other compiled with a PowerPC compiler.

Chapter 9 and 10 discuss differences in 680x0 code and PowerPC code. Chapter 9 focuses on numerics—the numerical data types that are used in Power Mac programming. Chapter 10 covers the steps needed to port existing 680x0 code to native PowerPC code.

Chapters 11 and 12 discuss import libraries. An import library—also called a shared library or dynamically linked library—allows a portion of an application to be saved as a library that can be used by multiple applications. This makes modification of code easy—the effects of a single change to an import library carry over to all applications that use that library. Chapter 11 shows that import libraries also make the creation of “plug-in tools” an easy task. A plug-in tool is a feature that can be distributed—or withheld—from a program. An example would be an application that, for an additional fee, comes with an import library that allows that application to play QuickTime movies. Chapter 12 discusses adding icons to import libraries, and introduces Apple events and their use in applications that work with import libraries.

Chapter 13 ends the book with a discussion of a few techniques for optimizing your PowerPC code. While source code may work without following the tips discussed in this chapter, it may not execute as fast and as efficiently as code that does.

WHAT'S ON THE DISK

The disk that comes bundled with this book has a single folder on it. Within that folder are three more folders—Metrowerks Examples *f*, Symantec Examples *f*, and Utilities *f*.

The Symantec Examples *f* holds the source code files and project files for the Symantec Cross-Development Kit (CDK) examples that are covered in this book. If you have the CDK, which is a Symantec product that can be used in conjunction with the Symantec 7.0 compiler, you'll find that everything is all set up for you.

The Metrowerks Examples *f* contains the source code files and project files for each of the Metrowerks examples discussed in this book. All of the Symantec examples are repeated here in CodeWarrior format—you won't have to make any changes to the source code or project files. If you have the Metrowerks compiler, you'll find that you'll save a lot of typing by using these projects.

The Utilities *f* folder contains a data fork erasing utility program named DFerase. You'll use this Macintosh utility in Chapter 8 when you turn a fat binary—a program that runs on both a 680x0-based Macintosh and a Power Macintosh—into a smaller program that runs only on Power Macs.

WHAT YOU NEED

To understand the contents of this book you should be familiar with a higher-level language—preferably C or C++. All source code listings are given in C. You should also be familiar with basic Macintosh programming concepts such as programming with the Toolbox.

All you need to run the example programs included on the disks is a PowerPC compiler. Either the Metrowerks CodeWarrior PPC compiler or Symantec's Cross-Development Kit (CDK). If you have either of these compilers you can compile all of the source code from either a 680x0-based

Programming the PowerPC

Macintosh or a Power Macintosh. The results of some of your compiles—the executables—will only run on a Power Mac, however. This is especially true of programs that use import libraries. Import libraries, or shared libraries, require calls to the Code Fragment Manager—a manager available only on Power Macs.

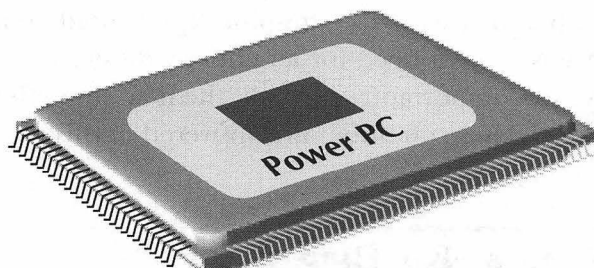
WHY THIS BOOK IS FOR YOU

Most programs that were written for a 680x0-based Macintosh will run, unchanged, on a Power Macintosh. So what's all the talk about PowerPC programming about? Here are a few reasons why any Macintosh programmer should be concerned about programming for the new PowerPC-based Macs:

- Programs designed for a 680x0-based Mac may run on a Power Mac, but they won't take advantage of the processing power of the PowerPC chip. The PowerPC executes native code—code that consists of PowerPC instructions—far more efficiently than it does 680x0 code.
- Some 680x0 code—especially code that use procedure pointers, or ProcPtrs—must be changed in order to execute on a Power Macintosh.
- Import libraries, or shared libraries, are dynamically linked libraries that are now fully supported on the Power Mac. They provide a powerful and easy way to make sections of your code modular and reusable.
- Apple has added to new important managers to the operating system—the Mixed Mode Manager and the Code Fragment Manager.

If you'd like to know about any of these topics, this book is for you. More generally speaking, if you've programmed the Mac, but aren't sure how to go about writing programs that run—and run fast—on a Power Mac, this book is for you.

Programming the PowerPC covers all of the above-mentioned topics, and several others. You'll find this book contains a background on the PowerPC chip architecture, a complete guide to porting 680x0 code to PowerPC code, a chapter on numeric data types for the Power Mac, and tips on optimizing your PowerPC code. There's plenty of example C language source code in the book—and on the included disk. And if you own either the Metrowerks CodeWarrior compiler or the Symantec Cross-Development Kit (CDK), you'll also find the disk contains project files all set up for your compiler.



CHAPTER 1

THE POWERPC AND THE POWER MACS

Apple's Power Macintosh computers—based on the new PowerPC microprocessor chip—were introduced with as much fanfare as the very first Macintosh computers back in 1984. Is all this publicity just industry hype—a gimmick to try to spark interest in still one more of the many new processor chips that have been developed over the years? Intel, the manufacturer of the chip that is inside computers that compete with the Macintosh, doesn't think so. They ran a succession of ads downplaying the PowerPC. Intel knows the potential of the chip that is the driving force of the new PowerPC Macs. This new Macintosh isn't a gimmick at all. Apple is betting its future on it—they expect to ship one million of the new PowerPC Macs in the first year alone.

In the last decade, improvements to the hardware and system software of the Mac have caused the Macintosh to evolve into the industry

Programming the PowerPC

standard for what an easy-to-use, graphically-oriented computer should be. Why then was there a need for an entirely different microprocessor? And, perhaps more importantly, how will the new chip effect how people perceive the Mac? These questions are answered in this chapter.

THE NEED FOR A NEW CHIP

In the ten years since its introduction, millions of Macintosh computers have been sold. In the past few years several very popular new models—such as the Quadra and PowerBook—have been introduced. All of these Macs are based on the Motorola 680x0 series of microprocessor chips. So why, as the Macintosh enters its second decade, is Apple scrapping the 680x0 chip for a new microprocessor? The answer is found in two acronyms, CISC and RISC, and the performance plateau one of them is reaching.



680x0 is sometimes written as 68 K. In either case, it's normally pronounced 68-kay. The original Macs were equipped with the 68000 chip. Since then Apple has used 68020, 68030, and 68040 chips as well.

CISC and RISC

Before the new PowerPC-based Macintoshes, all Macs used one of the Motorola 680x0 chips. The processor chips in this series use a *CISC*, or *complex instruction-set computer*, architecture. The first Macs, introduced a decade ago, contained the 68000 chip. Successive models contained the 68020, 68030, and finally the 68040 chip. Each new chip, and each Mac that housed a new chip, was faster than its predecessors. But improvement boosts between chips has started to level off, and engineers feel they've reached a plateau. They've pushed the CISC architecture to its limits and maximized chip performance. The Macintosh has outgrown the CISC chip, and must move on to something more powerful—enter RISC technology.

Chapter 1 The PowerPC and the Power Macs

CISC processors contain a wide variety of instructions meant to handle many different tasks. The new *RISC*—or *reduced instruction-set computer*—processors hold a much smaller number of instructions. Only the most basic, commonly used instructions exist in the instruction set of a RISC processor.

With far fewer instructions, how is a RISC processor capable of performing the same tasks as a CISC processor? RISC processors build complex instructions from combinations of their core of basic instructions. RISC processors are built in such a way that they execute basic instructions very fast—much faster than a more complex CISC instruction can execute. In RISC, when a more complicated task is required, a more complicated instruction is put together from the basic instructions. This offsets the performance gains that are made by running fast basic instruction—but only slightly. On a whole, the RISC chip runs faster than the CISC. Figure 1.1 illustrates the speed differences for CISC and RISC processors.

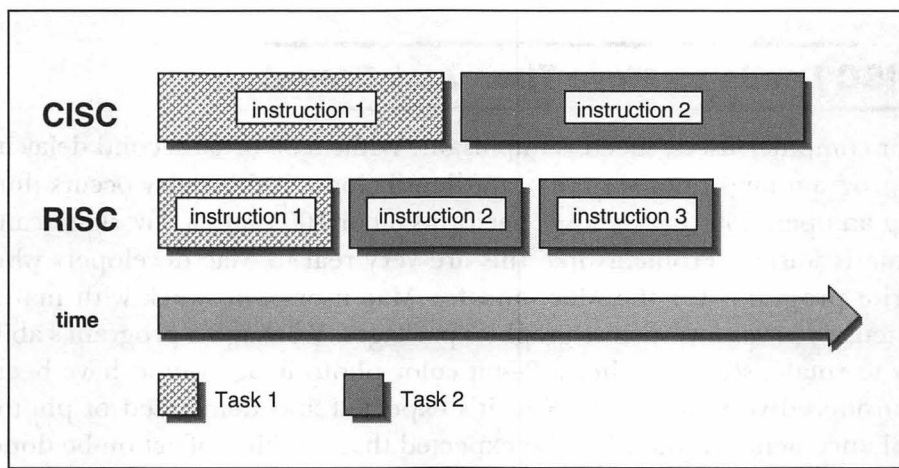


FIGURE 1.1 EXECUTION SPEED OF INSTRUCTIONS FOR CISC AND RISC PROCESSORS.

In Figure 1.1, two tasks are being carried out by both a CISC and a RISC processor. The CISC processor uses one complex instruction for each task.

Programming the PowerPC

The RISC processor uses a single basic instruction to carry out the first task. For the second task, it must build a more complex instruction from two basic instructions. Even with the building of a complex instruction, the overall time to execute the two tasks is still less for the RISC processor.



If engineers knew the CISC design was complex years ago, why didn't they change it long before now? The acronym CISC didn't exist until the advent of RISC. Before reduced instruction set computing, what is now called CISC wasn't thought of as complex.

The following two points summarize in a very general fashion the differences in architecture between CISC and RISC:

- CISC: many instructions, each specialized
- RISC: few instructions, each general

RISC Leads to More Than Just Speed

For computer users, speed is important. While a 30 or 60 second delay in a program might not seem extraordinarily long, if this delay occurs during an operation that the user performs 50 or 100 times a day, significant time is wasted. Problems like this are very real to Mac developers who write programs for the Mac, and for Mac users who work with math-intensive programs such as graphics packages. Years ago a program's ability to rotate, scale, or filter a 24-bit color photo image would have been considered very high-tech. Now it's expected and demanded of photo-enhancement software. It's also expected that this kind of action be done quickly. That's why the speed that accompanies the RISC architecture is so very important. Processor speed, however, isn't the only benefit that Apple hopes to gain by its switch to RISC.

Creative computing ideas often don't become reality, not because of a lack of technical know-how, but because of a lack of processor speed. Many of the exciting things being done on computer workstations aren't

being done on home computers simply due to a shortage of computing horsepower. Apple hopes—and expects—that programmers will use the extra speed of the PowerPC chip to develop programs with features and capabilities that would have been too much for 680x0 Macs. QuickTime's display of real-time video is an example of a technology that couldn't be supported by the original 68000 Macs, but was possible with the later and more powerful 68020 and higher chips. Apple hopes the muscle of the PowerPC will bring about similar advancements in video, graphics, communications, and areas not yet even imagined.

THE POWER MACINTOSH LINE

The new family of Macintosh computers—each based on the PowerPC microprocessor chip—is called the *Power Macintosh* line. The introduction of the new Power Macs includes three models—all based on the PowerPC 601 chip.

Features of the New Macs

The first three Power Macintosh computers are the 6100/60, the 7100/66, and the 8100/80. In recent years Apple has been criticized for its confusing array of model names and numbers—the Power Macintosh naming convention puts an end to that. The following pieces of information can be extracted from the model number:

- The higher the number, the more powerful the Mac
- The second digit describes the PowerPC chip in the Mac
- The number following the slash is the processor's clock speed

From the first point you know that the 6100 is the least powerful, the 8100 is the most powerful, and the 7100 lies in between the other two. From the second point you know that all three use the PowerPC 601 chip—the “1” in 6100, 7100, and 8100 stands for the 601 chip. The third

Programming the PowerPC

point tells you that the three computers run at 60 MHz, 66 MHz, and 80 MHz, respectively.

Table 1.1 gives an overview of the features of the first three Power Macintosh models that will be produced.

TABLE 1.1 OVERVIEW OF THE MODELS IN THE POWER MACINTOSH LINE.

			
Model	6100 / 60	7100 / 66	8100 / 80
Processor Chip Speed	PowerPC 601 60 MHz	PowerPC 601 66 MHz	PowerPC 601 80 MHz
RAM Standard Maximum	8 MB 72 MB	8 MB 136 MB	8 MB 264 MB
Expansion Slots	One 7" NuBus	3 full-size NuBus	3 full-size NuBus
Video DRAM video VRAM video VRAM maximum	Standard	Standard 1 MB 2 MB	Standard 2 MB 4 MB

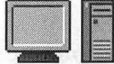
Each of the three Power Macs is available in a multimedia version. The model numbers are the same, only each is appended with an "AV." The 6100/60AV, 7100/66AV, and the 8100/80AV all include built-in video and frame capture and built-in sound and speech capabilities.

The meaning of most of the numbers in Table 1.1 should be intuitive, with the exception of the figures pertaining to video. 604 KB of DRAM, or dynamic RAM, is standard on all three models. It allows 8-bit color on Apple 16" monitors and 16-bit color on Apple 14" displays. An 8-bit color level means 256 colors can be displayed at any given time, while a 16-bit level means over 32,000 colors can be shown.

Chapter 1 The PowerPC and the Power Macs

Both the 7100 and the 8100 models supplement the standard DRAM with VRAM, or video RAM. The standard 1 MB of VRAM in the 7100 means that it can display 16-bit color on an Apple 16" monitor and 24-bit color on an Apple 14" screen. A 24-bit color level allows over 16 million colors to be displayed at any one time. The standard 2 MB of VRAM in the 8100 means it supports 24-bit color on both the 16" and 14" screens and 24-bit color on the Apple 19" and 21" monitors. Table 1.2 sums up the color capabilities of the Power Macs.

TABLE 1.2 COLOR CAPABILITIES OF THE THREE POWER MACINTOSH MODELS.

	 6100 / 60	 7100 / 66		 8100 / 80	
	Standard	Standard	Expanded	Standard	Expanded
12" Color					
14" Color					
13" VGA					
15" Portrait					
16" Color					
19" Color					
21" Color					



N/A



8-bit



16-bit



24-bit

In addition to the features already mentioned, each of the Power Macintosh models have 16-bit stereo input and output, on-board Ethernet, and the Apple Desktop Bus (ADB) for input devices.

The Customer Base

Apple envisions different sets of users for the different Power Mac models. The 6100/60 is the lowest priced of the three Power Macs, and is considered the entry level model. Its target customer base consists of the following groups:

- Small businesses
- Computer enthusiasts
- DOS/Windows users switching to Macintosh

The 7100/66 is the mid-range Power Macintosh model. Its faster speed and extra slots broaden its appeal to include the following groups:

- All businesses
- Education and administration
- Entry level professionals
- DOS/Windows users switching to Macintosh

The most powerful and expensive of the three Power Macs is the 8100/80. Its speed, multiple slots, and support of 24-bit color on a monitor up to 21 inches in size makes it ideal for the following markets:

- Professional publishing
- Engineers
- Multimedia authors
- DOS/Windows users switching to Macintosh

One group of users appears as part of the target audience for all three Power models—DOS and Windows users. All three of the Power Macs are capable of running Insignia Solutions' SoftWindows—emulation software that allows Macs to run DOS and Windows programs. That's appealing for users who are considering switching to Macintosh—it means they can get rid of their IBM-compatible hardware but retain their software investment.

Apple sees the 6100/60 appealing to DOS/Windows users because of its modest cost—something users of IBM clones are used to. The appeal of the 7100/66 is its three slots. Users of IBM clones are used to multiple slots and expect it in a computer. The 8100/80 will be the fastest of the Power Macs. That will appeal to the IBM compatible users who savor speed.

Is It STILL A MAC?

Making a dramatic change to the hardware that makes up a computer can provide that machine with a new appeal to many people. It can also scare off many potential computer buyers who fear struggling with issues such as learning a new system, loss of investment in 680x0 software, incompatible hardware add-ons, and the learning curve associated with programming a new computer. Apple kept all of these issues in the forefront as it implemented the hardware architecture of the new Power Macintosh computers.

The PowerPC System Software

In moving from a CISC microprocessor to one that employs RISC technology, Apple invested a large amount of time and money in changing the internal workings of the Macintosh. This was done because of the belief that the CISC technology of the Motorola 680x0 series was reaching its maximum potential. Apple did not, however, feel the graphical user interface that is the trademark of the Macintosh was also showing signs of aging. So the windows, menus, and icons that define the Mac have not changed.

The Power Macs use a version of System 7—so they have the exact same user interface as the 680x0 Macs. The first version of System 7 that supports the Power computers is System 7.1.2.

Programming the PowerPC

Software Compatibility

Owners of 680x0 Macs typically have hundreds or even thousands of dollars worth of software. For users moving up to a Power Mac, this sizable investment is not lost. All software that was properly designed to run on a 68020, 68030, or 68040 Macintosh will run on a Power Macintosh.

Most major software vendors have modified their software programs that were originally designed to run on 680x0 Macs so that they will run much faster on Power Macs. Most older programs that were not modified will also run on the new Macs—they just won't take full advantage of the speed of the PowerPC chip.

Hardware Compatibility

Users of the Power Macs may see their new computer as exciting, blazingly fast, and fun to work with. Hardware components connected to a Power Mac will see the new computer as just another Macæand that's good news. It means that 680x0 printers and SCSI devices such as hard disks and scanners can be used with the new Macs.

For businesses, hardware compatibility means Power Macs can be added to an existing network of 680x0 Macs without confusing the server. The Power Macs can run old and new programs and transfer files between other Power Macs and older 680x0 models.

Developer Support

Programmers who have grown accustomed to developing for the Macintosh have fears about programming the Power Macintoshes. For them, there is more good news. Writing software that will run on both the older 680x0 Macs and the new Power Macs can be done with a minimum of effort. Programmers do not need to learn an entirely new programming language or learn all the details of a new operating system. Greater effort is needed to take full advantage of the processing speed of the new Macs—but not an unreasonable amount.

Chapter 1 The PowerPC and the Power Macs

This book deals with PowerPC issues that concern Mac programmers:

- porting existing 680x0 source code to run on the PowerPC,
- writing new code that maximizes the power of the PowerPC,
- taking advantages of programming techniques unique to the PowerPC, and
- 680x0/PowerPC compatibility issues.

CHAPTER SUMMARY

Before development of the PowerPC chip, all Macs used a microprocessor from the Motorola 680x0 family of chips. These 680x0 chips used a CISC, or complex instruction-set computer, architecture. The new PowerPC chips that are the driving force of the Power Macs use a RISC, or reduced instruction-set computer, architecture. RISC technology is superior because it uses a smaller set of instructions than CISC. From these fewer, simpler instructions, the RISC chip can carry out simple tasks quicker than a CISC chip. And for more complicated tasks, the RISC chip can string together a series of simple instructions that still run more quickly than the complex instructions of a CISC chip.

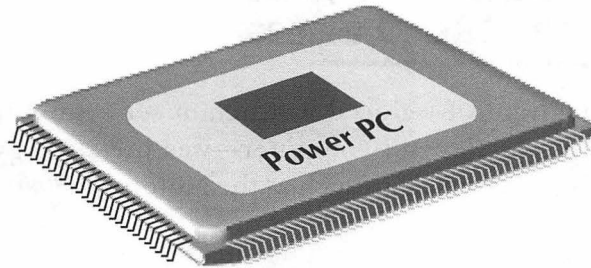
There are three models in the first series of Macintosh computers that use the PowerPC chip. The 6100/60 is the low-end Power Mac aimed at small businesses and computer enthusiasts. The 7100/66 is the midrange model, and is aimed at businesses of all sizes, educational users, and entry-level professionals. The 8100/80 is the most powerful of the three models. Its appeal will be to professional publishers, engineers, and multimedia authors. Because the Power Macs can run DOS and Windows programs, all three models are ideal for users who are making a transition from the IBM-compatible world.

The technology behind the change from a 680x0-based processor to a PowerPC-based processor will be transparent to end-users. The Macintosh system software used by the Power Macs appears identical to that used by 680x0 Macs. And, though older versions of existing pro-

Programming the PowerPC

grams won't take full advantage of the speed of the PowerPC chip, they will nonetheless run on the new Macs.

For programmers, getting old Macintosh code to compile and run on Power Macs will not be a daunting task. But there are several programming issues programmers should be aware of. With the PowerPC there are also new features that developers will want to consider adding to new programs—such as import libraries. The remainder of this book will discuss these points at length.



CHAPTER 2

CISC AND RISC TECHNOLOGIES

Though the PowerPC is a new microprocessor, the RISC technology that it is based on came into existence over fifteen years ago. As RISC evolved, it both borrowed from CISC and diverged from it into completely new directions. An understanding of the Motorola 680x0 series, and the CISC technology upon which it is based, will help in understanding the newer RISC technology used in the PowerPC chip.

While the Power Macintoshes are the computers everyone is talking about, Macs based on the 680x0 series will be around for years to come. Anyone designing software for the Power Macs will want to ensure that their programs also are compatible with the millions of 680x0 Macs currently on the market. This chapter discusses how both the 680x0 and the PowerPC work with instructions. This information will serve as a background for future chapters that discuss writing source code that is compatible on both old and new Macintoshes.

Programming the PowerPC

CISC AND THE 680x0 SERIES

Before the advent of reduced instruction set computing, CISC—or complex instruction set computer—was the strategy used in the design of microprocessor chips such as the Motorola 680x0 series.

Why CISC?

The CISC design evolved from two sets of circumstances that prevailed in the world of computers years ago:

- Computer programs were written primarily in assembly language
- Computer memory was relatively slow and expensive

The CISC design uses methodologies that aid assembly language programmers and reduce access to the slow, expensive, main memory chips. By making each instruction perform multiple tasks, assembly language programmers were spared the necessity of learning large sets of instructions. And by having the instructions perform much of the work within registers in the processor, long, slow trips to main memory were avoided.

Instruction Execution on a 680x0

The 680x0 processors have a number of general purpose registers for holding data and addresses, as well as a few special purpose registers. One of these special registers—the Status register—is used to hold information about the status of the Mac. The Status register is divided into two parts. One part—the CCR, or condition code register—is of more significance to programmers than the other part. The CCR holds information about the outcome of the most recently performed arithmetic or comparative operation. The CCR thus gives a programmer information such as whether or not an operation resulted in a negative value or a value of zero. Figure 2.1 shows many of the registers in a typical 680x0 chip.

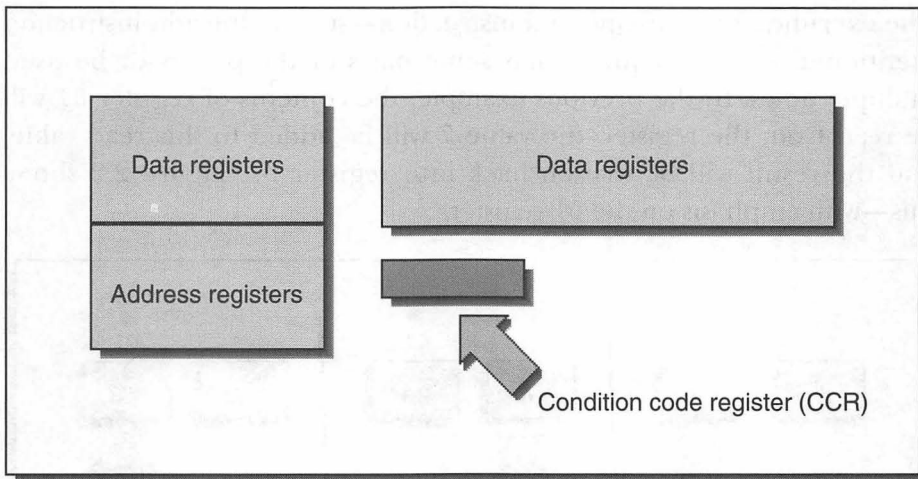


FIGURE 2.1 SOME OF THE REGISTERS IN A 680x0 SERIES CHIP.



NOTE

Though this section discusses registers and shows a couple of assembly language instructions, a knowledge of assembly language is not required for comprehension of the topics presented here.

Like any microprocessor, the 680x0 carries out instructions by performing arithmetic operations on data. That data may be moved between main memory and registers within the processor, or from register to register within the processor. The rich instruction set of a 680x0 microprocessor holds instructions that accept 0, 1, or 2 operands. An operand holds data—or the memory address of data—that is to be acted on.

An example of an assembly instruction that uses two operands is the add instruction. It adds two operands together and stores the results back in one of the two operands. Here's an example that adds the value 7 to the contents of register D1 and stores the result back in D1:

```
ADD #7,D1
```

Programming the PowerPC

The execution of a two-operand instruction—such as the add instruction mentioned above—requires that some parts of the processor be used multiple times. In the previous example, the contents of register D1 will be read from the register, the value 7 will be added to this read value, and the result will be written back into register D1. Figure 2.2 shows this—with emphasis on the D1 register.

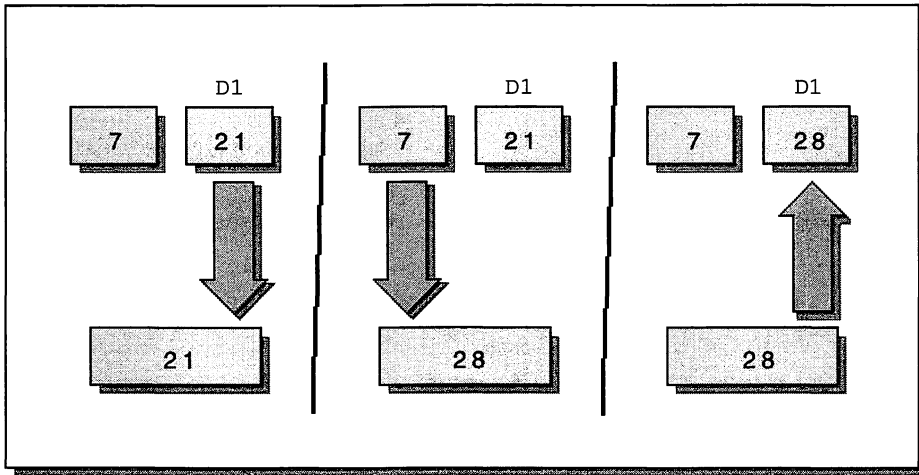


FIGURE 2.2 INSTRUCTION EXECUTION CAN USE ONE REGISTER MORE THAN ONCE.

After the completion of the add instruction the condition code register—the CCR—will hold information about the addition operation that just took place. In fact, most assembly instructions affect the condition codes in this register.

This discussion has shown that two-operand instructions use and reuse registers and that the condition code register is usually altered after the execution of an instruction. These two concepts are very important limiting factors in the determination of the speed at which a 680x0—or any other CISC microprocessor—can process instructions.

Back in Figure 2.2, you saw that the example add instruction took more than one step to complete. Of the three steps shown, the first and last both involve the D1 register. That means that at any point in time

between these steps, the D1 register cannot be used by another instruction. If another instruction were to start executing before the add was complete (and it too used the D1 register) it might interfere with results obtained during the add. For this reason, any 680x0 processor, and any other CISC processor, cannot start executing a new instruction until the currently executing instruction has completed.

Once an instruction is complete, it sets various flags in the condition code register. The next instruction that executes may examine these flags and use information from one or more of them. This is a second reason that CISC processors can't run concurrent instruction—an instruction must complete before the condition code register flags get set.

The lack of the ability to start executing an instruction until the currently running instruction completes is a key factor that restricts the speed that a CISC processor can obtain. RISC processors use a design implementation that overcomes this boundary—as you'll see later in this chapter.

The Timing of Instructions on a 680x0

While a 680x0 processor can't execute more than one instruction at a time, it can perform more than one task at a time. Typically, a single instruction is divided into multiple stages that run consecutively. The result of one stage can be used by another stage.

Let's look at a hypothetical example that demonstrates the multiple stages of an instruction. Assume there exists an instruction that draws a square in a window. To take advantage of the Mac's graphics capabilities, the instruction tries to give the square a three-dimensional look by first drawing a black square, then a white square slightly offset from the black one. The result of one execution of this instruction might look something like that shown in the window in Figure 2.3.

One execution of this square-drawing instruction takes one clock cycle. Now, to get a better feel for the timing involved as a CISC processor executes instructions, imagine that a program has been written that repeatedly uses this instruction. Not too sophisticated a program, but its

Programming the PowerPC

simplicity and repetitiveness will serve well to demonstrate the timing of instructions in a CISC processor. Figure 2.4 shows the timing and outcome of the program after four clock cycles have completed.

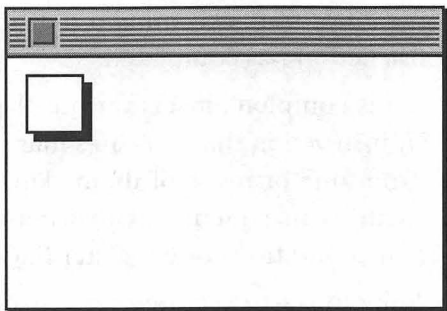


FIGURE 2.3 RESULTS OF A HYPOTHETICAL SQUARE-DRAWING INSTRUCTION.

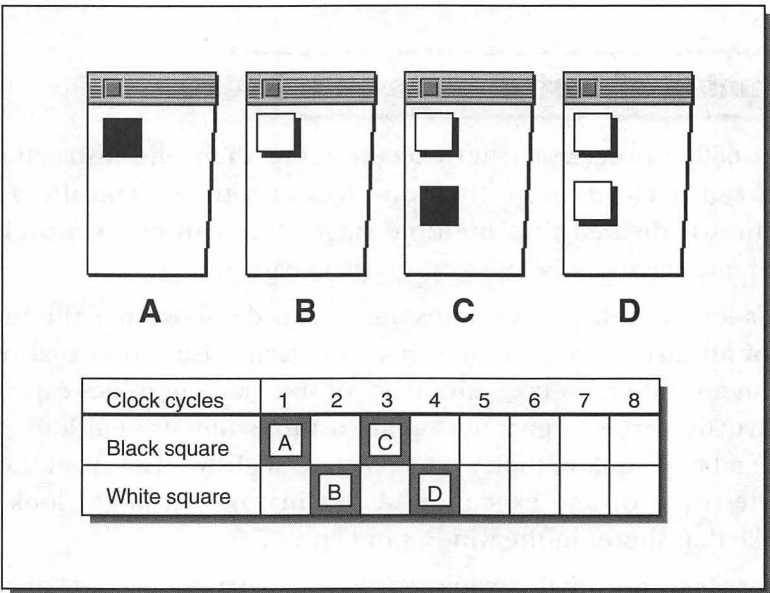


FIGURE 2.4 INSTRUCTION TIMING IN A HYPOTHETICAL CISC PROGRAM.

The task of the first clock cycle is the drawing of the black square—the result is shown in part A of Figure 2.4. The second and final task of the instruction is to draw the white square over the black square, as shown in part B of the figure. The third clock cycle starts the second execution of the square-drawing instruction. By part D of the figure, the instruction—and the fourth clock cycle—are complete.

Figure 2.5 shows another way of looking at the instruction timing. This figure places the emphasis on the objects—the two squares. This figure shows that it takes two cycles to create a single object. After four cycles, two objects have been created. It also clearly illustrates that a second instruction will not begin until the first instruction has completed.

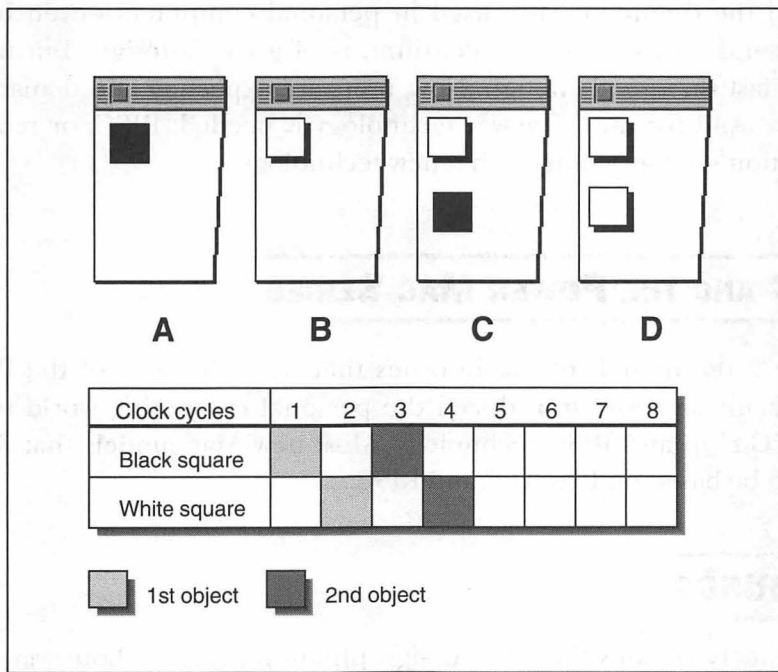


FIGURE 2.5 TIMING OF THE DRAWING OF AN OBJECT IN A HYPOTHETICAL CISC PROGRAM.

Programming the PowerPC



Important! If you take nothing more away from this section than the idea that the CISC chip is incapable of starting one instruction before completing the previous one, you've learned enough!

CISC—Fast, But Not Fast Enough

This section has emphasized the pitfall inherent in complex instruction set computing—the inability to run concurrent instructions. But that's not to say that CISC is slow. A CISC chip like the Motorola 68040 makes software on the Quadra Macintosh models run very quickly. CISC chips are still the dominant chip used in personal computers—even Intel's newest and fastest chip, the Pentium, is of a CISC design. But in the 1990's, fast isn't good enough. Now, users are expecting and demanding *very* fast. And for that, a newer technology is needed. RISC, or reduced instruction set computing, is that new technology.

RISC AND THE POWER MAC SERIES

The three models of Macintoshes that were the start of the Power Macintosh series introduced the personal computing world to the PowerPC chip and RISC technology. Most new Mac models that follow will also be based on PowerPC and RISC.

Why RISC?

The primary reasons the CISC design philosophy came about were that programmers used assembly language and memory was slow and expensive. Today, assembly language is a dying art. Even programs that are dependent on great speed are usually written in a higher-level language, with a few assembly routines added to handle the most speed-intensive

operations. This, coupled with the fact that computer memory has become very fast and very inexpensive, has greatly reduced the benefits of a chip design based on the CISC philosophy.

CISC chips are complex—that's a result of the complexity of the instructions that the chip can execute. For the benefit of assembly language programmers, CISC chip instructions usually perform multiple tasks. The hardware necessary to make this come about is thus more complex. And complex means slow. A chip designed to carry out only simpler instructions will inherently run faster.

CISC uses powerful instructions that perform numerous tasks to make life easier for assembly programmers. In recent years compilers have been optimized so that they can take code written in a high-level language such as C or C++ and generate machine language code that is as fast and efficient as code written in assembly language. Because of this, the use of assembly language has been diminishing while the use of high-level languages has soared. Developers who write programs in an "English-like" higher-level language need not memorize obscure assembly mnemonics. For the architects of processor chips, that meant that they could look at new ways of implementing instructions within a chip. From this reexamination of instruction execution, RISC technology was born.

Instruction Execution on a Power Mac

The PowerPC chip, like the 680x0 chip, has internal general purpose and special purpose registers. One of these is the CR, or condition register—it's analogous to the 680x0 CCR, or condition code register. And, again like the 680x0 chips, some PowerPC instructions require the reuse of one or more of the internal registers. Yet the RISC technology in the PowerPC chip overcomes this obstacle so that the execution of one instruction can begin before the completion of another.

In CISC, instructions vary in length and complexity. A single instruction may require more than one fetch—more than one access of main memory to obtain instruction information. In RISC, all instructions are of the same length. Each instruction can be fetched in a single opera-

Programming the PowerPC

tion. And almost all RISC instructions are so basic that they can be completed in a single cycle.

While all RISC instructions are simple, all of the tasks expected of a microprocessor are not. How does the PowerPC handle an operation that is of greater complexity than any one RISC instruction is capable of performing? In such an instance the PowerPC will execute a series of simple instructions to handle this one more complex task.

It might seem like the total amount of time to execute a series of simple instructions could be equal—or even greater—than the time to run one complex instruction. If the simple instructions were run one after the other, with no overlap in execution, this indeed would be the case. But in RISC, one instruction can begin before another ends. This technique of building a complex instruction from several basic instructions results in a net savings of time.

In order to take advantage of the RISC chip's ability to overlap instruction execution, a stumbling block present in CISC technology had to be overcome—instruction dependencies. If a second instruction depends on the results produced by a first instruction, the execution of the second instruction can't begin until the completion of the first. To circumvent this problem the PowerPC actually rearranges the instructions found in a program to reduce or eliminate dependencies. This technique is called *instruction scheduling*.



The rearrangement of instructions is obviously not a haphazard effort. A programmer has some control over this process, but compilers such as Metrowerks' CodeWarrior and Symantec's Cross Development Kit (CDK) do the majority of the work in determining where dependencies lie, and where they can safely be removed. Additionally, the PowerPC chip itself uses sophisticated logic to determine the ordering of instruction execution.

Note that as a programmer you need not be concerned about the instruction rearranging that your compiler and Power Mac institute. Only instruction switching and dependency removals that don't harm the integrity of data will be

performed. If a rearrangement of instructions would lead to an incorrect instruction result, the switch will not be made.

Let's take a look at an example that demonstrates how instruction scheduling might work. For the 680x0, the add instruction has two operands. The value in the first operand is added to the second, and the result is stored back in the location named as the first operand. For the PowerPC, the instruction to perform addition has three operands. The second and third operands specify the values to add, while the first operand specifies where the result should be stored. Here's an example that adds the number seven to the contents of register r2, then stores the result back in r2:

```
add r2, r2, #7
```

Now, here's the above instruction included in a very short snippet of code:

```
add r2, r2, #7
stw r2, total
sub r3, r3, #5
```

In the above example, the value seven is added to the contents of register r2 and the result is stored back in r2. Then, the stw instruction stores this sum into the word in memory addressed by total. Finally, a subtraction, or sub, instruction performs a subtraction using register r3.

In previous sections you've seen that the add instruction reuses the register that serves as both the source of one of the values to add and the destination of the final result. In the above example, you see a similar situation—register r2 will be accessed twice. While the addition operation is taking place, no other instruction that uses the r2 register can run. Since the instruction that follows the add—the stw instruction—uses r2, it cannot begin running until the add has completed. The add must complete so that r2 contains the final and correct value that the stw instruction is to write to memory.

You know that a RISC system attempts to start one instruction while running another one. When the system has to wait for an instruction to

Programming the PowerPC

complete before beginning the next instruction, a stall results. Instruction scheduling eliminates stalls. In the next code snippet the order of the second and third instructions of the previous example have been switched:

```
add r2, r2, #7
sub r3, r3, #5
stw r2, total
```

This one simple change prevents a stall. While the add instruction is running, the sub can begin. Why? Because there is no instruction dependency—the sub instruction does not use r2. Because the sub instruction doesn't make use of r2 or access the memory location addressed by total, whether the result of the add is stored in memory after the first instruction or after the second instruction is unimportant. Figure 2.6 illustrates.

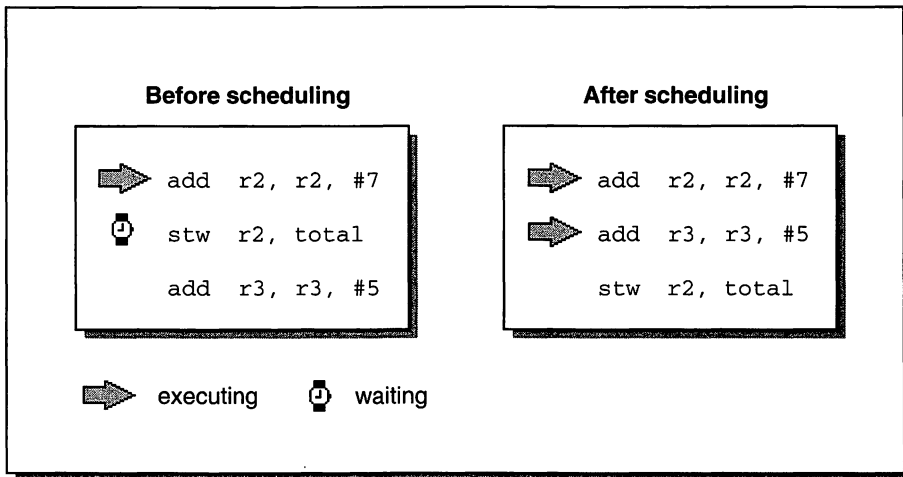


FIGURE 2.6 A EXAMPLE OF INSTRUCTION SCHEDULING ELIMINATING A STALL.

In a PowerPC, the switching of the order of instructions is commonplace. This is a feature of the PowerPC chip—as a programmer, you don't have to use special programming techniques to achieve instruction scheduling.

The Timing of Instructions on a Power Mac

The goal of RISC is to speed up processing. To do that, the PowerPC employs a technique not possible in a CISC chip—*pipelining*. Pipelining is the processing of one instruction before a previous instruction has completed. Hardware design considerations in the PowerPC, along with the reduction of instruction dependencies through instruction scheduling make pipelining possible.

You've seen that in a CISC system instructions are of varying lengths—depending on the complexity of the task the instruction is to perform. In RISC, each instruction is the same length. For complex operations, several instructions are joined to build one larger instruction. Thus in either CISC or RISC, the completion of a complex task can take more than one cycle. The RISC technique of pipelining doesn't decrease instruction *latency*—the number of cycles to complete a single instruction. It does, however, increase *throughput*—the number of instructions completed per cycle. That's because instructions are broken into many tasks—just as in CISC. But unlike CISC, the RISC processor doesn't wait for one instruction to end before starting another. Thus several instructions can run concurrently and the flow of instructions speeds up.

As an example of instruction timing in a RISC system, refer to Figure 2.7. Here you see the execution of the same square-drawing instruction used in the CISC example earlier in this chapter. Once again I'll request that you cast aside your doubts as to the usefulness of a program that mindlessly draws square after square!

The first clock cycle of the RISC system is identical to that of the CISC system—a black square gets drawn. But in the second cycle, things have changed. With pipelining, the RISC processor finishes the first square-drawing instruction by drawing the white square over the black one. But it also starts the second square-drawing instruction. So in the second cycle both a white square and a black square are drawn. The third cycle completes the second square and starts the third. The fourth cycle completes the third square and starts a fourth square.

Programming the PowerPC

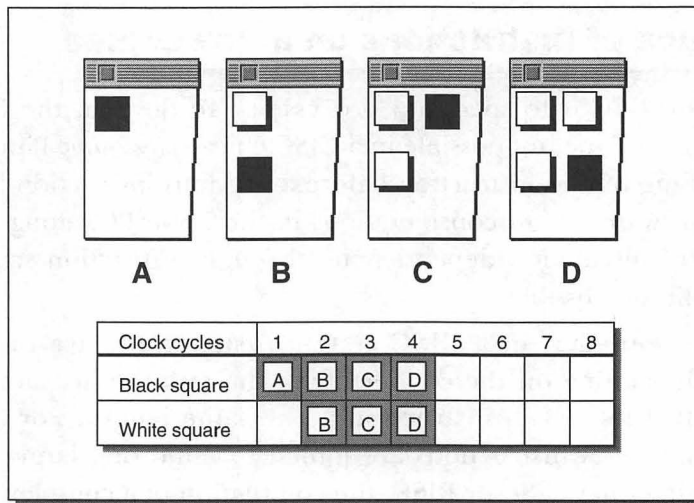


FIGURE 2.7 INSTRUCTION TIMING IN A HYPOTHETICAL RISC PROGRAM.

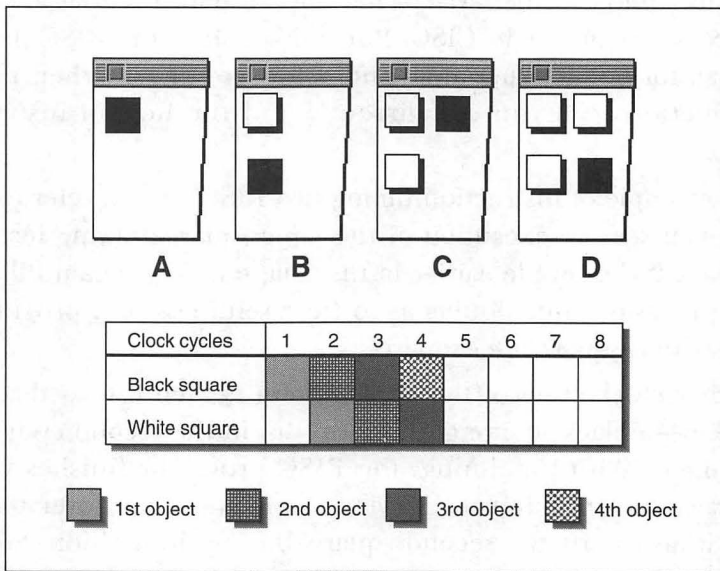


FIGURE 2.8 TIMING OF THE DRAWING OF AN OBJECT IN A HYPOTHETICAL RISC PROGRAM.

Figure 2.8 takes another look at the pipelining of instructions. Here, emphasis is on the objects.

Note that while it takes two stages to create each individual object—just as it did in the CISC system—after the first stage a new object is created every stage. After four cycles, three objects have been created and a fourth has been started. Contrast this RISC result from that obtained with the CISC processor. In this chapter's CISC example only two objects were created in four cycles. Pipelining allows the RISC processor to start on a new object before the previous object is complete. Figure 2.9 contrasts the difference in instruction processing for CISC and RISC systems.

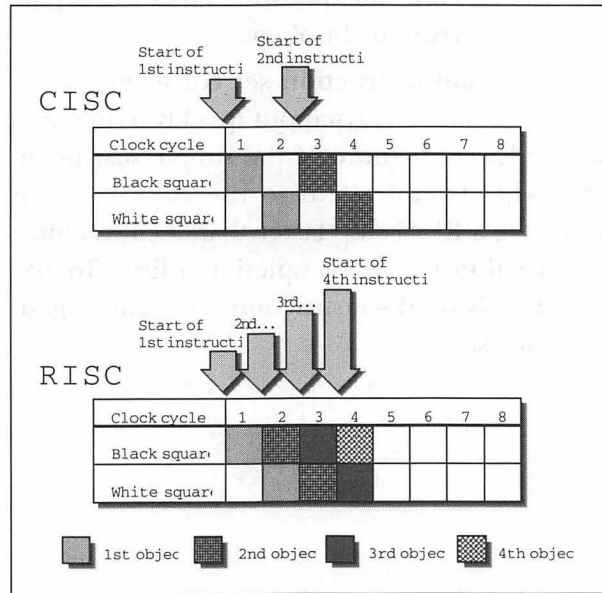


FIGURE 2.9 TIMING COMPARISON OF CISC AND RISC PROCESSORS.

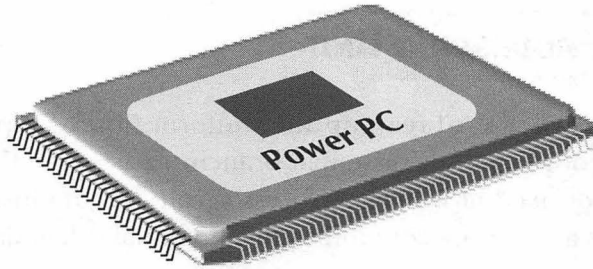
CHAPTER SUMMARY

CISC—or complex instruction set computer—is a design that uses methodologies that aid assembly language programmers and reduce

access to the slow main memory chips. The rich instruction set of a 680x0 microprocessor holds instructions that accept 0, 1, or 2 operands. Two-operand instructions use and reuse registers, and alter the value in the CPU's condition code register after the execution of an instruction. These two concepts are important factors that limit the speed at which a CISC microprocessor can process instructions. On a CISC processor, one instruction must complete before a second instruction can start.

CISC held advantages over some technologies when assembly language programming was prevalent and memory was slow. Today, high-level languages are far more popular than assembly language. Also, main memory has become fast and inexpensive. These facts pave the way for RISC to become a mainstream technology.

RISC relies on a small instruction set consisting of simple instructions. Each instruction can be carried out quickly. When a more complex instruction is needed, two or more of the simple instructions are strung together to form what is equivalent to the complex instruction. The other factor in making a RISC chip faster than a CISC chip is its ability to start a second instruction before completing a first. To accomplish this, instruction scheduling is used—instructions are rearranged to reduce or eliminate dependencies.



CHAPTER 3

POWERPC ARCHITECTURE

In Chapter 2 you saw that pipelining—the ability to start executing an instruction before another has completed—is the key to the speed of the PowerPC microprocessor. In this chapter you'll see the chip architecture that makes pipelining possible.

BRANCH PROCESSING UNIT

Branch instructions break up the uniform flow of a program. So it should come as no surprise that branches are one of the key limiting factors in the speed at which a processor can execute instructions. The PowerPC has a hardware solution to this potential dilemma—the Branch Processing Unit.

Instruction Fetching

The PowerPC, like any microprocessor, runs a program by fetching an instruction from RAM and then executing that instruction. Figure 3.1 shows this unending cycle of fetching and executing instructions. How the PowerPC executes any one instruction depends on the type of instruction fetched—so for that reason the components that make up the PowerPC chip were intentionally left unlabeled in Figure 3.1.

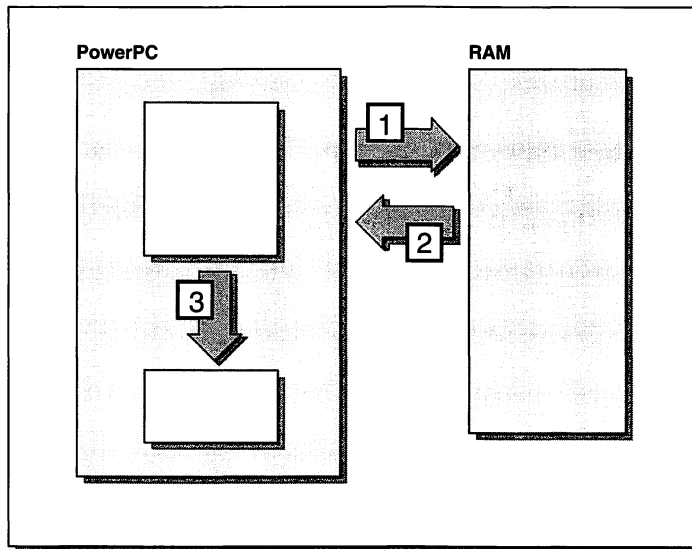


FIGURE 3.1 THE PROCESSOR CONTINUOUSLY FETCHES AND EXECUTES INSTRUCTIONS.

The PowerPC is fast—very fast. At times it may execute instructions faster than RAM can deliver new ones. To avoid a pause in instruction execution, the PowerPC employs instruction *prefetching*. Prefetching involves fetching several instructions at one time, and loading those instructions into a queue on board the PowerPC. During lulls in the delivery of instructions, the PowerPC can continue to dispense queued instructions to the appropriate components for processing. Figure 3.2 shows the instruction queue in place in the PowerPC.

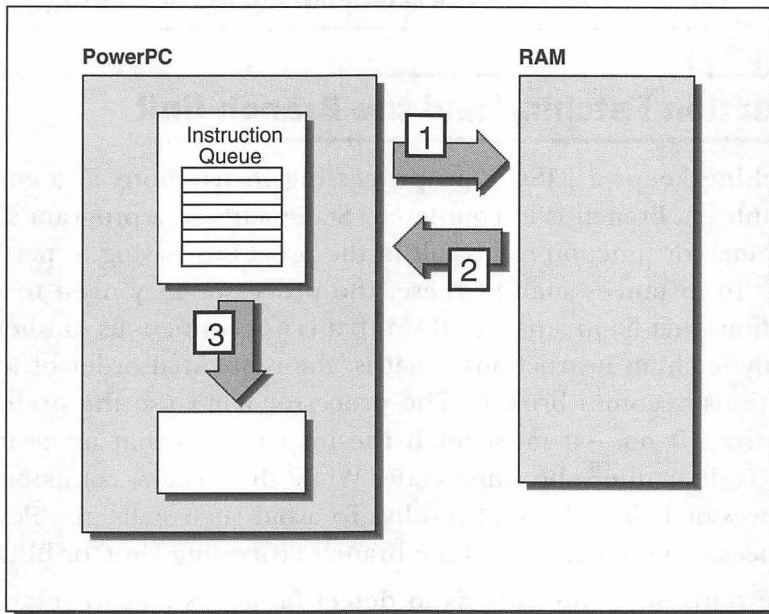


FIGURE 3.2 FETCHING INSTRUCTIONS TO THE INSTRUCTION QUEUE.



Of interest to programmers who use assembly language is the new bit numbering convention of the PowerPC. When working with a 32-bit word, the 680x0 series calls the right-hand bit “bit 0” and the left-hand bit “bit 31.” On the PowerPC, bit numbering is reversed. This difference is shown in Figure 3.3.

Programming the PowerPC

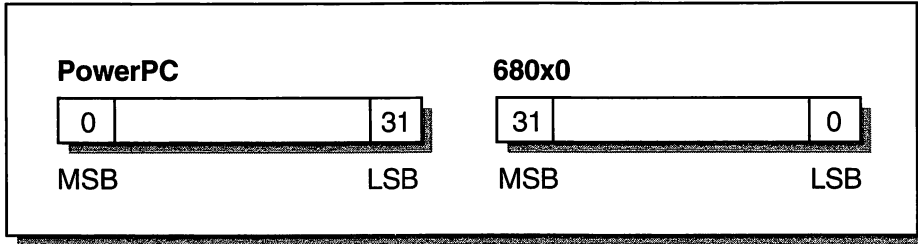


FIGURE 3.3 BIT NUMBERING IN THE POWERPC AND THE 680x0.

Instruction Fetching and the Branch Unit

Prefetching keeps a RISC chip processing instructions at a constant rate—unless a branch is encountered. Statements in a program such as an *if* branch or function call result in the processor having to perform a branch. In instances such as these, the processor may need to access instructions that lie in a part of RAM that is not contiguous to where it is currently fetching instructions. That is, the sequential order of fetched instructions becomes broken. The processor can't use the prefetched instructions it has—it must fetch the instructions that make up the branch code or the subroutine code. While this occurs, components of the processor halt and a stall results. To avoid such stalls, the PowerPC has a special component called the Branch Processing Unit, or BPU.

The purpose of the BPU is to detect branches and to respond by making adjustments to the instruction queue. Figure 3.4 shows that the BPU, along with the instruction queue, make up a component called the Instruction Unit. The PowerPC chip of course contains more units than the Instruction Unit—the unlabeled boxes in Figure 3.4 hint at that.

The Branch Processing Unit constantly examines the last few instructions in the instruction queue. If a branch is included in the instructions, the BPU will fetch the appropriate instructions based on the branch. Figure 3.5 shows a few lines from a program written in C language. Each line is numbered, and the line numbers of a few of the lines are shown in the instruction. This is done for the sake of clarity—a real PowerPC

Chapter 3 PowerPC Architecture

would be working with the machine language that resulted from the source code, not the source code itself. Figure 3.5 shows how you might expect the instruction queue to look after the PowerPC encountered the first three lines of code. In fact, the instruction queue would look like that pictured in the following figure, Figure 3.6.

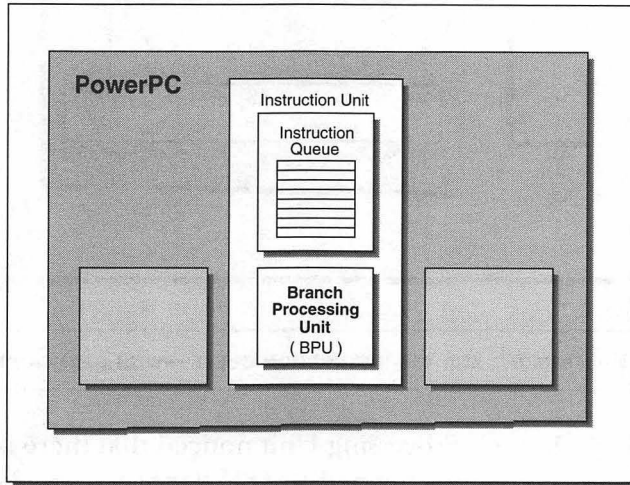


FIGURE 3.4 THE BRANCH PROCESSING UNIT COMPONENT OF THE POWERPC CHIP.

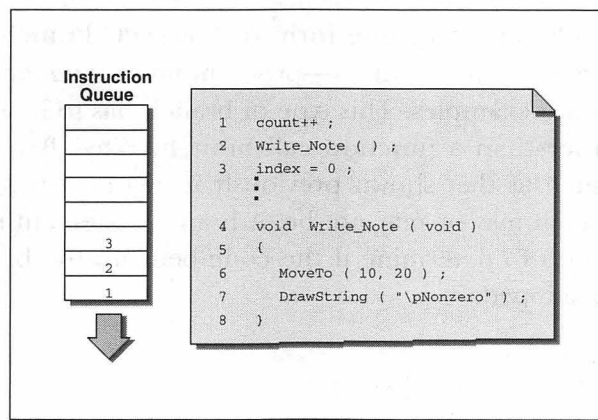


FIGURE 3.5 PREFETCHING AND THE INSTRUCTION QUEUE—WITHOUT BRANCH PROCESSING.

Programming the PowerPC

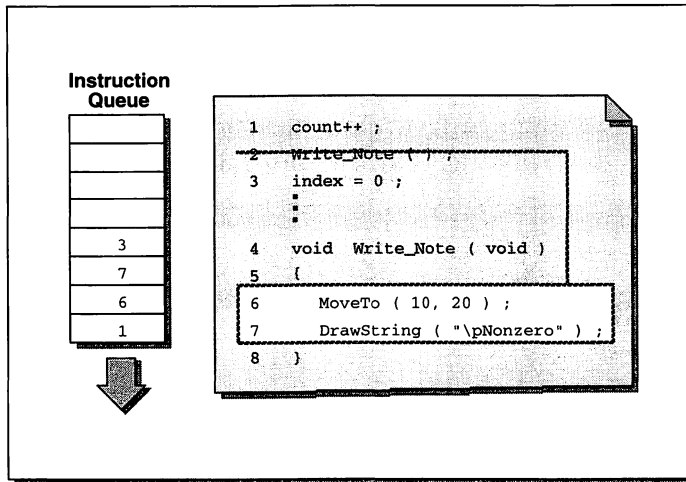


FIGURE 3.6 PREFETCHING AND THE INSTRUCTION QUEUE—WITH BRANCH PROCESSING.

In Figure 3.6, the Branch Processing Unit noticed that there is a function call in the code—a function named `Write_Note()` is invoked. A function call is a branch. The BPU then replaces the function call with the instructions that make up the function itself. In this case the function consists of two Toolbox routines—`MoveTo()` and `DrawString()`.

Function calls represent one form of PowerPC branch. The other kind is an actual branch statement—programming statements such as *if*, *switch*, and *goto* are examples. This type of branch has to be handled in a different manner than a function call branch. Why? Because while a function call such as that shown previously in Figure 3-6 will always be executed, a branch may or may not be. A branch statement relies on the test of a condition to determine if the code beneath the branch should be executed or skipped:

```
if ( index > 0 )
    DrawString( "\pNonzero" );
```

The course that will be taken by a branch such as this is determined by conditions at runtime. At the time that the Branch Processing Unit

examines this section of code, the value of index might not be known. So, as the BPU examines a branch statement for which it doesn't know the outcome, how does it determine which instructions to place in the instruction queue? It doesn't. But it still places the instructions that make up one body of the branch into the queue.

The BPU uses *branch prediction* to make a decision regarding an unresolved branch. A guess is made as to which path will be taken, and the instructions that make up that path are prefetched and placed in the instruction queue. If the prediction is correct, instruction processing executes without a stall—that is, without delay. The PowerPC hedges its bet by storing the address of the nonpredicted path. That way, if the prediction proves to be incorrect, the instructions that make up the other path of the branch can be fetched.

It's important that the BPU guess correctly much more often than not—an incorrect prediction obviously negates any time savings that would have resulted from a correct prediction. To aid in the determination of the correct branch instructions to prefetch, the BPU follows a set of fixed rules. One such rule is that the *if* path of a branch will always be selected over the *else* path.

Examination of source code shows that programmers will usually (intentionally or not) place the code that is most likely to execute under the *if* rather than the *else*. Imagine a program that contains an integer variable named *index*. The program allows this variable to be assigned any value between 0 and 100. If the program is interested only in whether *index* is either zero or in the range of 1 to 99, a programmer will usually write the code as shown here:

```
if ( index > 0 )
    DrawString( "\pNonzero" );    /* 99 of 100 end up here */
else
    DrawString( "\pZero" );       /* 1 of 100 end up here */
```

Again taking the liberty of using source code for clarity, Figure 3.7 shows an example of how the BPU would start to fill the instruction queue as it encounters a section of code that includes an if-else statement.

Programming the PowerPC

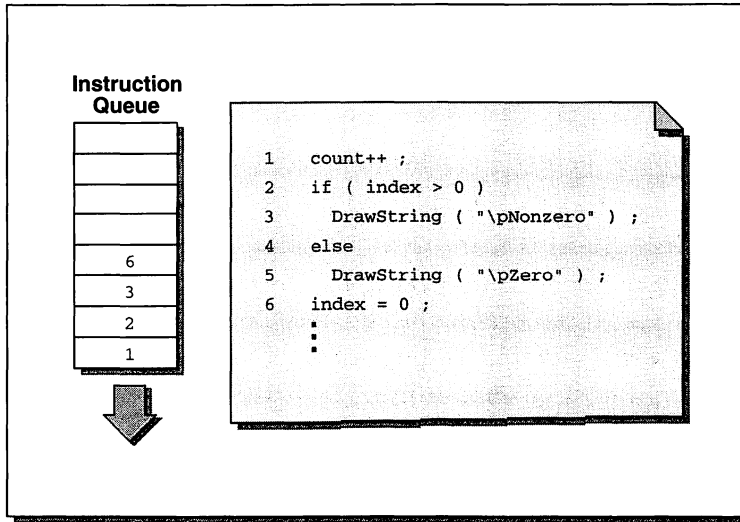


FIGURE 3.7 BRANCH PREDICTION IS USED TO RESOLVE BRANCHING.



NOTE

Branch prediction is another good reason to use a compiler designed to optimize code for the PowerPC—as the Metrowerks and Symantec CDK compilers do. These compilers are set to correctly handle conditional branches.

SUPERSCALING

Simplifying each instruction in a chip's instruction set is one way to reduce the execution time of instructions. You've seen that the PowerPC does just that. A second means of speeding up instruction execution is to run instructions concurrently. The PowerPC employs a *superscalar* design to accomplish this feat.

The Superscalar Design

Parallel processing is the use of multiple processors to speed up instruction execution. Another method used to achieve this effect is to design the parts of a single processor such that they run in parallel to one another. This design strategy is called *superscalar design*, and is the one used by the PowerPC.

The PowerPC relies on three functional units that work both independently and in conjunction with one another to execute instructions. You are already familiar with one of the units—the Branch Processing Unit. The other two units are the Integer Unit, or IU, and the Floating-Point Unit, or FPU. Figure 3.8 shows the three major components of the PowerPC.

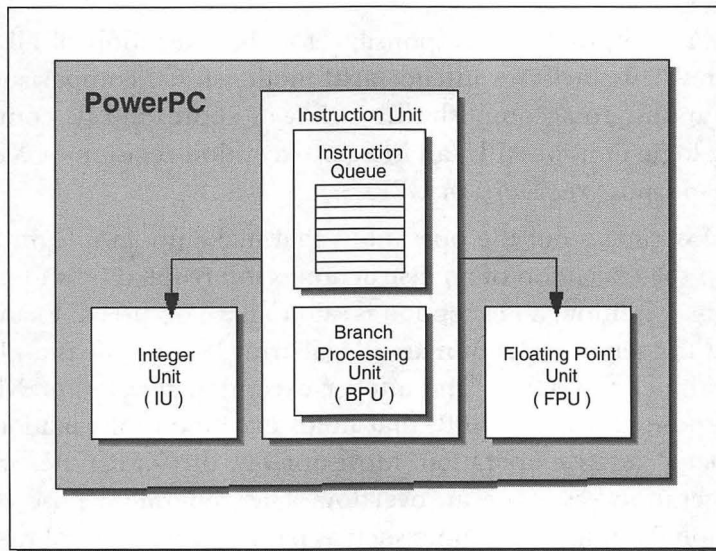


FIGURE 3.8 MAJOR COMPONENTS OF THE POWERPC CHIP.

Having three separate units means that at any one time the PowerPC can execute three separate instructions: an integer instruction, a floating-point instruction, and a branch instruction.

Branch Processing Unit

The Branch Processing Unit, or BPU, was covered earlier in this chapter. Its purpose is to locate branches before they occur. Then, using branch prediction, the BPU prefetches what it believes are the instructions that make up the branch.

The BPU is located, along with the instruction queue, in the instruction unit. That's because all instructions pass through the BPU first. That enables the BPU to examine each instruction for a branch. Instructions that aren't branches are then handled by either the Integer Unit or the Floating-Point Unit.

Integer Unit

The Integer Unit, or IU, is responsible for the execution of all integer instructions. This includes integer arithmetic, shifts, comparisons, and logic operations. To accomplish this variety of chores the IU contains an arithmetic logic unit, or ALU, an integer exception register, or XER, and 32 general-purpose registers, or GPRs.

The ALU carries out the operations that make up an integer instruction. When the execution of an instruction is interrupted by an unexpected or illegal condition, an exception is said to have occurred. Examples of events that trigger an exception are illegal array bounds, division by zero, and an arithmetic overflow. The integer exception register, or XER, is a special-purpose register, or SPR, that holds exception information about the completed integer operation. Most notably, the XER notes when an integer operation results in an overflow. The general-purpose registers hold any non-floating-point values such as parameters and local variables.

Floating-Point Unit

The Floating-Point Unit, or FPU, is responsible for the handling of all instructions that involve floating-point values. This includes floating-point arithmetic operations and compares.

The FPU contains parts that are roughly comparable to those in the Integer Unit. Because arithmetic operations for floating-point values use different logic than those performed on integers, these operations are performed by a multiply-add array rather than an ALU.

The one special-purpose register, or SPR, in the FPU is the floating-point status and control register—the FPSCR. The FPSCR holds exception information—such as overflow and zero divide status—about FPU operations, as well as the type of result produced by floating-point operations.

In the FPU, general-purpose registers are called floating-point registers, or FPRs. The FPU contains thirty-two 64-bit FPRs. These registers are used for purposes such as the passing of floating-point parameters.

CACHE

Cache memory is memory that is accessed more quickly than standard RAM. Computers usually offer the option of an add-on cache board to speed up the system—and the Power Macs are no exception. But the PowerPC also contains an on-board cache built into the hardware of the microprocessor chip itself. This cache actually consists of two separate areas—a data cache and an instruction cache with a combined size of 32K of fast memory.

Data Cache

The data cache holds the data of a program that is used the most often. Before accessing data from RAM, the PowerPC first checks to see if the referenced data is already present in the cache. If it is, the cache data is accessed rather than the external RAM data.

For the data cache, the PowerPC maintains cache *coherency*. A coherent cache means that the PowerPC will attempt to keep a synchronization between cache data and RAM data—over the next few pages you'll see how this works. To aid in this task the PowerPC maintains something called *snooping logic* that monitors bus addresses and makes comparisons

Programming the PowerPC

between these addresses and those in the cache. Without coherency, the data in the cache could change while the copy in RAM would remain the same—making the RAM value invalid.

In Figure 3.9, data from RAM is written to the PowerPC data cache. To remind you of the complexity of the PowerPC, a few of the several units are shown as gray boxes.

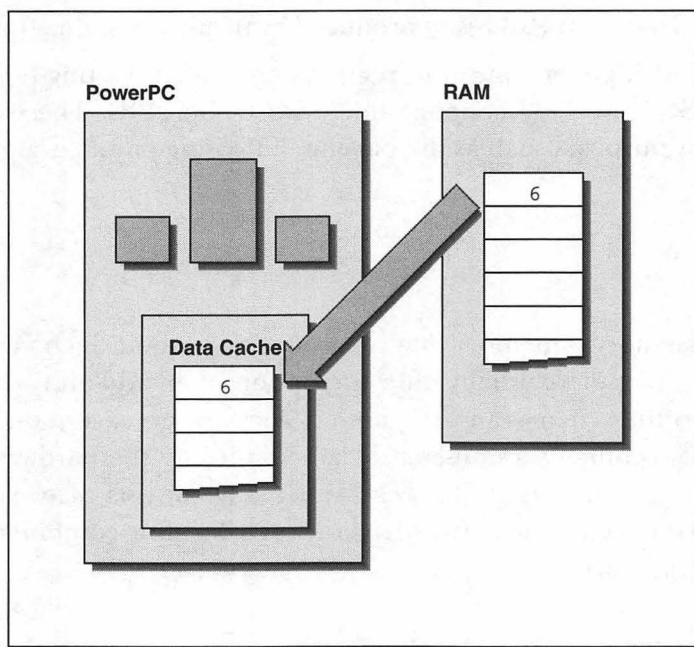


FIGURE 3.9 DATA FROM RAM GETS WRITTEN TO THE POWERPC DATA CACHE.

When a running application attempts to make use of some data, the PowerPC will first check the Data Cache to see if that data resides there. If it does, the Data Cache value will be read and sent to the program. This saves a trip to the slower external RAM. Figure 3.10 shows cache memory being accessed.

If the program that uses the data makes a change to it and then writes the new value to memory, that value will be first written back to the

fast Data Cache. Then, while the processor works on the next instructions, the newly-written value will be written back to RAM to keep the RAM value in sync with the cache value. This scenario is shown in Figure 3.11.

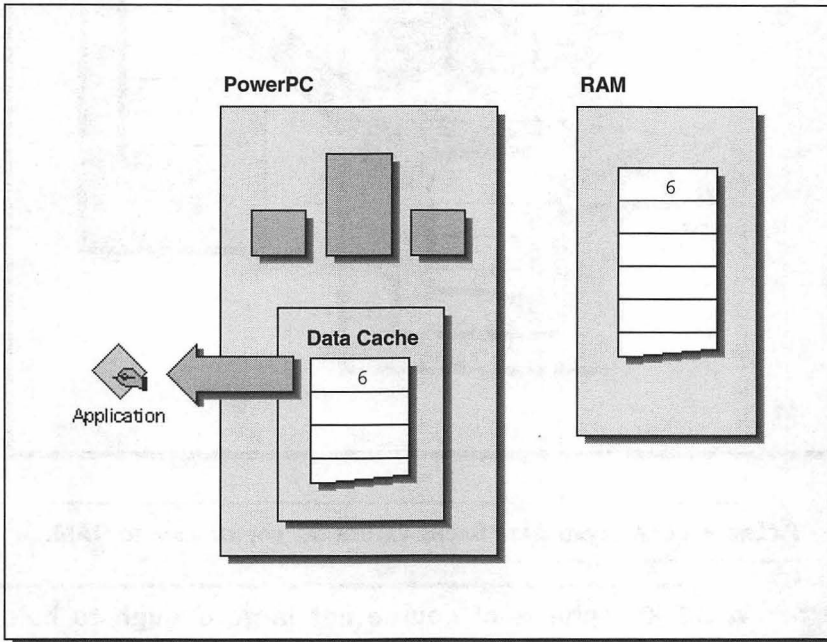


FIGURE 3.10 AN APPLICATION USES DATA FROM THE DATA CACHE.

Instruction Cache

The part of the PowerPC cache that holds instructions is *not* coherent. That means that the values in the cache will not always match the values in RAM. Coherency is not maintained for the Instruction Cache because the PowerPC makes the assumption that instructions are read-only and won't be modified. Once an instruction is read into the cache from RAM, it is thus assumed that this cache instruction will not change.

Programming the PowerPC

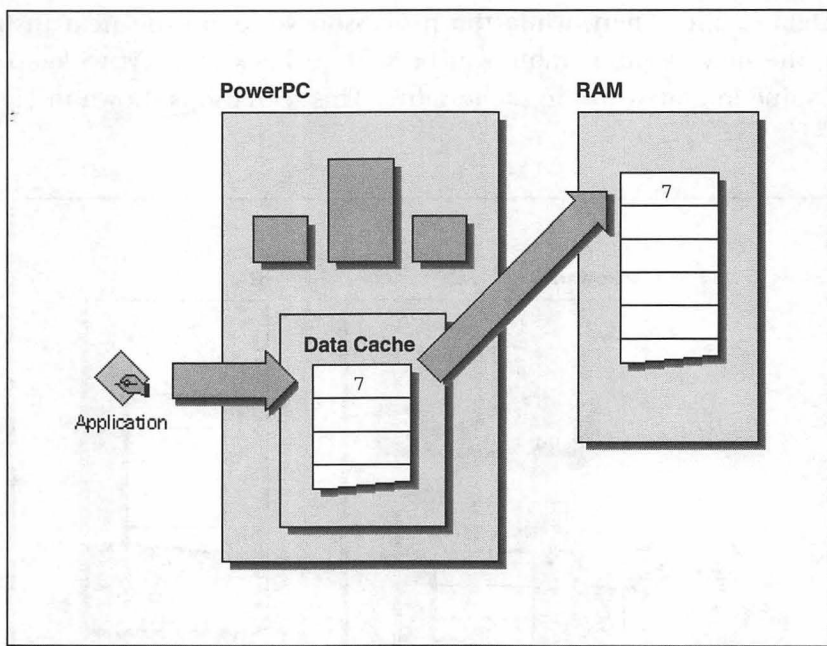
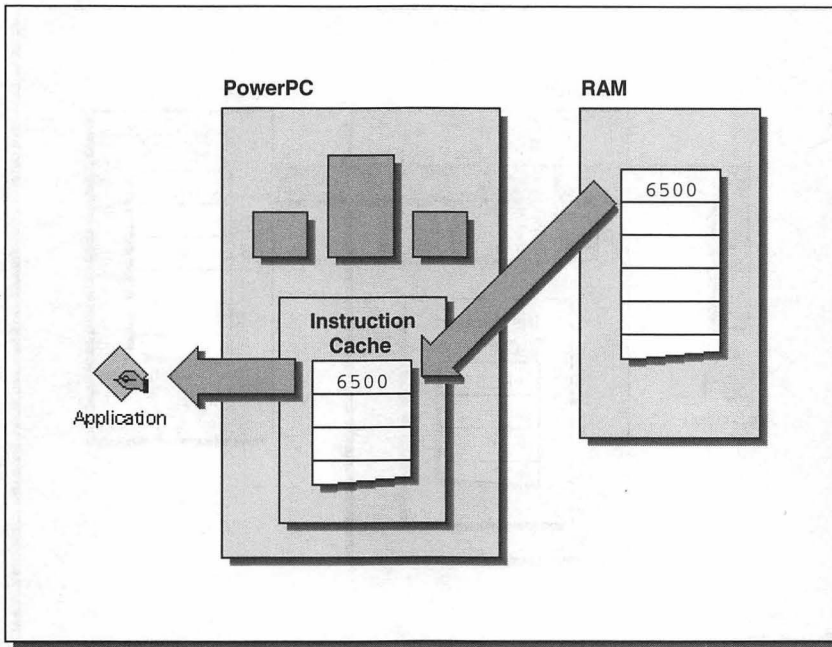


FIGURE 3.11 ALTERED DATA CACHE VALUES GET COPIED BACK TO RAM.



A 32 K cache is of course not large enough to hold an entire program. The cache is meant to hold heavily used data and instructions. If the cache is full, and some of its contents have not been used recently, new data and instructions will be moved in from RAM. When it's said that an instruction in the cache will not change, it's meant that once an instruction is read into the cache from RAM, that version of that instruction in the cache will not be altered. If the instruction is not readily used again, the cache location that holds it will eventually be overwritten by a new instruction from RAM.

Figure 3.12 shows an instruction being read from RAM and stored in the cache. The figure then shows that instruction being used by the running program. The program accesses the cache rather than main memory.



**FIGURE 3.12 AN INSTRUCTION FROM RAM
GETS WRITTEN TO THE POWERPC INSTRUCTION CACHE.**

If a program does change an instruction and writes it back to RAM (something that is not recommended), the cache will be bypassed. The program will write directly to main memory while the cache maintains the original instruction. Figure 3.13 shows this unlikely scenario.

CHAPTER SUMMARY

Pipelining—the ability to start the execution of a second instruction before a first instruction has completed—is key to the speed of the PowerPC chip. The architecture of the PowerPC chip makes pipelining possible.

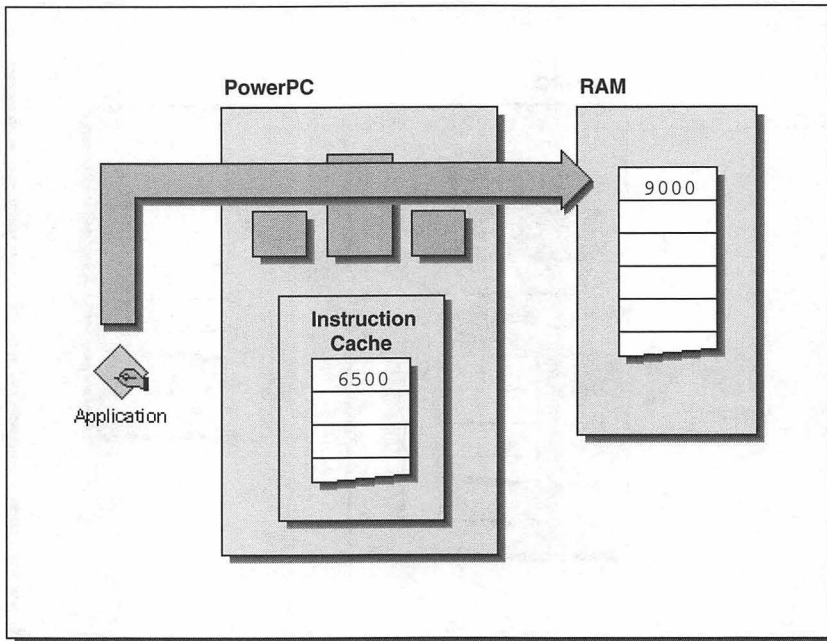
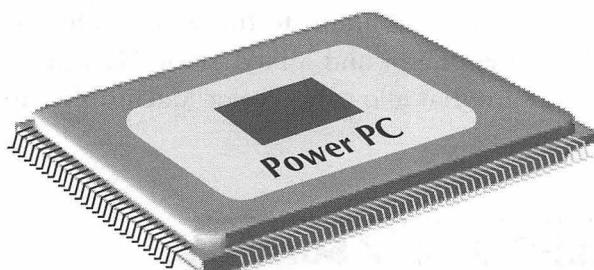


FIGURE 3.13 AN ALTERED INSTRUCTION BYPASSES THE INSTRUCTION CACHE AND IS INSTEAD WRITTEN TO RAM.

Branch instructions break up the uniform flow of a program, and thus have the potential to slow down a program. To overcome this dilemma, the PowerPC has a hardware solution called the Branch Processing Unit, or BPU. The BPU detects branches before the executing program encounters them. It then makes adjustments to the instructions queue—the holding place of prefetched instructions.

To further increase instruction execution speed, the PowerPC uses a design strategy called superscaler design. The PowerPC uses three functional units to execute instructions. The BPU, the Integer Unit (IU), and the Floating-Point Unit (FPU) work both independently and in conjunction with one another.

The PowerPC contains an on-board cache to speed up memory access. This 32K cache is composed of two separate areas—a data cache and an instruction cache.



CHAPTER 4

POWERPC SYSTEM SOFTWARE: THE EMULATOR AND MIXED MODE

The blazing speed of the PowerPC microprocessor—and of the Power Macintoshes based on it—will be the impetus behind the development of many new, exciting Macintosh software programs. In the near future, Mac users will see the unveiling of programs that couldn't exist without the processing muscle of the PowerPC. The reality for most users of Power Macintosh computers, however, lies in the full disk case that sits near their new machine. It houses an investment of hundreds or thousands of dollars in Mac software—software designed to run on 680x0-based machines. These users will be grateful to hear that as Apple planned for the future, it did not forget the present.

The new Power Macintosh computers are capable of running just about every existing Macintosh application—including those developed for the 680x0 Macs. Apple made this possible not through the PowerPC

Programming the PowerPC

chip, but instead through additions to the system software. This chapter covers the 68LC040 Emulator and Mixed Mode Manager—the new parts of the system software that allow the Power Macintoshes to run both old and new software.

THE POWERPC SYSTEM SOFTWARE

The software investment of 680x0 users is one reason that Apple wants to make sure the new Macs run 680x0 software. Here's a few other reasons:

- New Power Macs will be on networks along with older 680x0 Macs.
- Many people will be working with both old and new Macs and will need to run software that works on both.
- Independent software developers with access to only one type of machine will be developing for both the 680x0 and the Power Mac.
- New machines that don't have an abundance of software choices fair poorly.

Ported System Software Routines

Because of the design of the PowerPC chip, code runs quicker on a PowerPC-based Mac than it does on most 680x0-based machines. Further speed gains are noticed when code is recompiled using a compiler designed to optimize code for the PowerPC. This applies to all source code—yours and Apple's. Don't forget, your programs make extensive use of Toolbox and Operating System functions. Before ending up in ROM and in the System file, these routines started out as source code—source code that was originally written for 680x0 machines.

To get the most impressive speed increase from the PowerPC chip, all of the system software functions should be recompiled to make them

native PowerPC. Apple is in the process of doing that now. For the 6100, 7100, and 8100 Power Macs, about 10 percent of the Toolbox has been rewritten in C and recompiled into native PowerPC code.



You'll see the word "native" throughout this book, and throughout any PowerPC literature you read. *Native code* is code written specifically for a PowerPC-based Macintosh. A *native application* is an application that has been compiled or recompiled for the PowerPC microprocessor. An existing 680x0 application that runs on a Power Mac is not said to be native. Once appropriate changes are made to the application's source code, and the source code has been recompiled using a PowerPC compiler, the resulting application is then said to be a native program. Native applications take advantage of the speed increases offered by the PowerPC chip.

Ten percent of the system software might not seem significant. But before you feel slighted by this seemingly low figure, you should consider what Apple calls the "90/10 rule." Apple engineers examined the code of numerous programs and found that 90 percent of the calls to Toolbox functions are made to just 10 percent of the Toolbox routines. That means that most programs repeatedly call a handful of Toolbox routines and seldom (or never) call the vast majority of the thousands of routines. Figure 4.1 gives emphasis to this fact.

In particular, the `DrawText()`, `EraseRgn()`, `Line()`, and `GetFontInfo()` routines represented about two-thirds of all system calls. Though a program might not call these routines directly, other system functions may. For the first Power Macintoshes, these routines, the entire Memory Manager, QuickDraw, and some other calls were ported to native PowerPC code. The current Macintosh system software used by the Power Macs is thus a mixture of old 680x0 functions and new, native PowerPC functions. Over time Apple will port more and more of the system routines to native PowerPC code. As new system software is released (in the form of new System files), more and more of it will be native PowerPC. Figure 4.2 shows that both old and new routines exist in the system software.

Programming the PowerPC

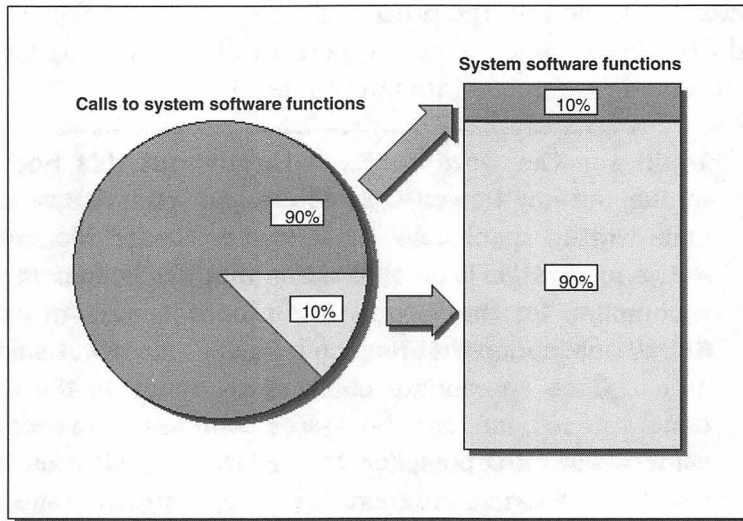


FIGURE 4.1 NINETY PERCENT OF SYSTEM CALLS ARE TO JUST TEN PERCENT OF THE SYSTEM FUNCTIONS.

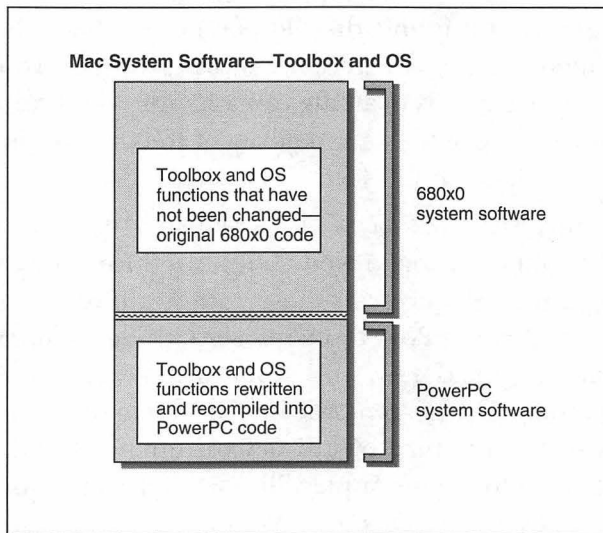


FIGURE 4.2 SYSTEM SOFTWARE IS A MIX OF 680x0 AND POWERPC ROUTINES.



As Apple ports more Toolbox routines to native PowerPC code, new versions of the System file will be released. These new versions will hold the new, faster Toolbox functions. That way owners of what will become “older” Power Macintoshes will be able to take advantage of the faster Toolbox by simply getting a new version of the system software.

The New System Software

A mixture of 680x0 routines and native PowerPC routines in the same Toolbox could represent a programmer’s nightmare—if it weren’t for a lot of help from two new additions to the system software. To support software running in a *mixed environment*, the Power Macs contain new ROM chips and a new System file that together contain the Mixed Mode Manager and the 68LC040 Emulator.

The first system software for PowerPC-based Macs is System 7.1.2. Apple wants the transition from 680x0 Macs to Power Macs to be smooth, with a minimum of new-technology shock for the user. For that reason, the screen of a Power Mac running System 7.1.2 will look like the screen of a 680x0 Mac running System 7.1. Inside the System, however, there are many new changes—the most notable of which is the Mixed Mode Manager.



The Power Macs won’t run using *any* version of System 6.x. The first Power Mac system software, 7.1.2, is obviously a version of System 7. For this reason, any software you write that is to run on a Power Mac, or both a Power Mac and a 680x0, must be compatible with System 7. That means your software should be 32-bit clean—as described in *Inside Macintosh Volume VI* and the latest versions of *Inside Macintosh*. Software should also be able to run in a multitasking environment and be compatible with the operations of the Virtual Memory Manager.

Programming the PowerPC

Like the other managers, such as the Memory Manager and the Window Manager, the Mixed Mode Manager is a set of functions that serves a common purpose. The purpose of the Mixed Mode Manager routines is to allow software to run in the mixed environment of 680x0 and PowerPC system software routines. The Mixed Mode Manager allows older programs designed for 680x0 Macs and new programs designed for the PowerPC to run side by side on a Power Macintosh.

A program designed for a 680x0—but is now running on a Power Mac—will not always take advantage of the speed of the PowerPC chip. One reason for this is that the program would not have been compiled using a compiler that optimized the code for the PowerPC processor. Another reason is that many of the Toolbox calls made by the program will be to 680x0 Toolbox routines. The PowerPC microprocessor does not recognize the older instructions that make up these 680x0 Toolbox routines. Since the Toolbox of the Power Macs contain a mix of old and new routines, how does the PowerPC chip housed in a new Mac resolve this dilemma? Through software emulation.

Any program that calls a 680x0 system software routine will find itself in the 68LC040 Emulator of the Power Mac. The Emulator is a software program that resides in the ROM chips of each Power Mac. Its sole purpose is to convert 680x0 instructions to PowerPC instructions and then pass these converted commands to the PowerPC chip for execution. Not all calls to the Toolbox are routed to the 68LC040 Emulator—only calls to the older 680x0 functions. And the mechanism responsible for sending a call to the Emulator? The Mixed Mode Manager.

PowerPC Execution of System Software Routines

The Mixed Mode Manager and the 68LC040 Emulator provide the Power Mac with the ability to run both 680x0 software and PowerPC software. Figure 4.3 gives an overview of the system software of a Power Mac.

Over the next several pages you'll see how a Power Macintosh handles each of the four combinations of calls that applications running on a Power Mac can make.

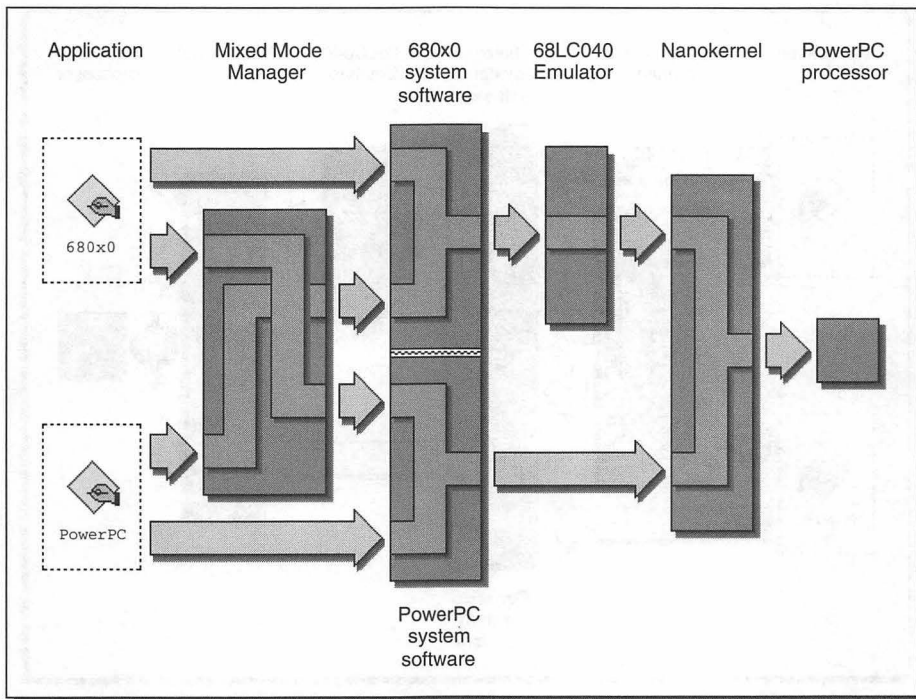


FIGURE 4.3 THE POWERPC SYSTEM SOFTWARE INCLUDES A MIXED MODE MANAGER AND A 68LC040 EMULATOR.

Let's first examine the case of a 680x0 application running unmodified on a Power Mac. If this program makes a call to a 680x0 Toolbox or Operating System routine, the call will go to the system software and then will be processed by the 68LC040 Emulator. The Emulator converts the instructions that make up this function call to PowerPC instructions. The Emulator then passes the processed instructions on to a low-level piece of system software called the *nanokernel*. The nanokernel communicates with the PowerPC chip and handles interrupts and memory management tasks. All calls, regardless of the path taken, pass through the nanokernel before reaching the PowerPC chip. Figure 4.4 shows the path taken by a 680x0 call made from a 680x0 application.

Programming the PowerPC

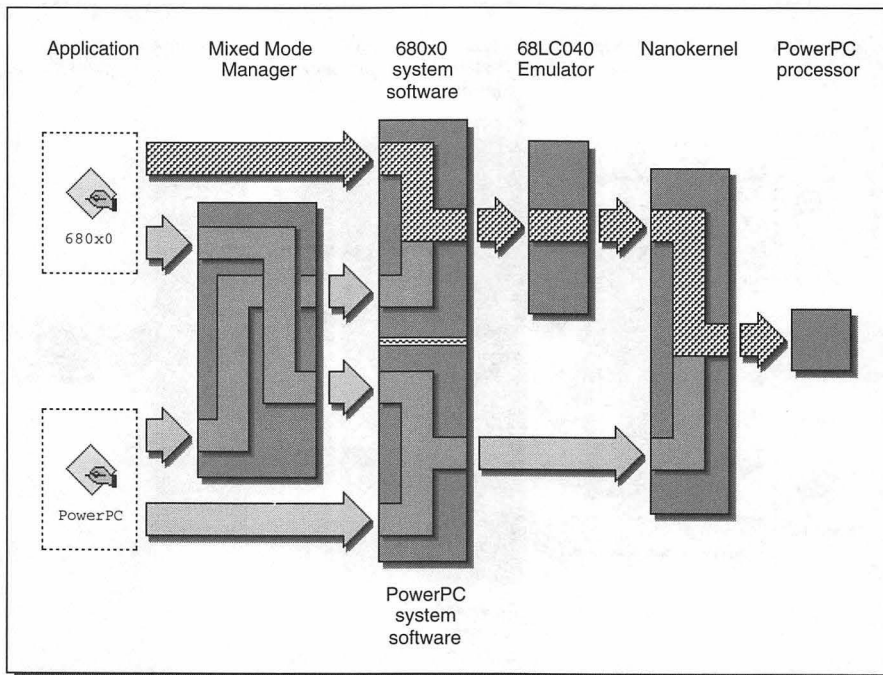


FIGURE 4.4 A 680x0 APPLICATION CALLING A 680x0 SYSTEM ROUTINE

You might guess that applications written before the Power Macs came into existence don't call PowerPC system routines—but in fact, they do. That comes about from the porting of some system routines—as was discussed in this chapter. Approximately 10 percent of the system calls were rewritten and made native PowerPC routines. If a 680x0 application calls one of these routines, such as `DrawText()`, then it is calling a PowerPC Toolbox function. In an instance such as this a *mode switch* occurs. The Power Mac switches from handling a call via the 68LC040 Emulator to handling it directly by the PowerPC. Since the Toolbox routine has been ported to native PowerPC code, there is no reason to translate the routine's instructions to PowerPC code. As Figure 4.5 shows, the Mixed Mode Manager is responsible for routing this kind of call to the PowerPC—without going through the Emulator.

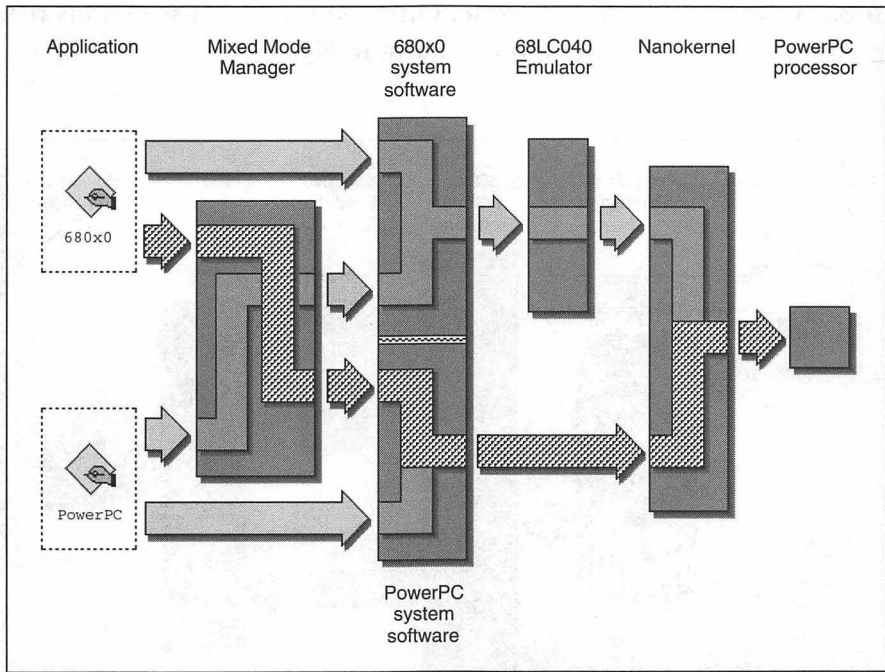


FIGURE 4.5 A 680x0 APPLICATION CALLING A NATIVE POWERPC SYSTEM ROUTINE.



NOTE

It's important that you realize that a program doesn't necessarily run in just one mode or the other. A typical program will be switching modes constantly throughout its execution.

Just as a 680x0 application can call both old and new system software routines, so can a PowerPC application. If a PowerPC program makes a call to a PowerPC Toolbox routine, the call is handled by the PowerPC processor with no instruction conversion. Figure 4.6 shows this situation.

As mentioned, the entire Toolbox has not been ported to native PowerPC code. That means that on occasion a new PowerPC program will have to make a call to an older 680x0 system routine. When that occurs, the Mixed Mode Manager becomes involved. It sends the call to the 68LC040 Emulator so that the instructions that make up the routine

Programming the PowerPC

can be converted to PowerPC code. Only then do the instructions reach the PowerPC chip. This situation is shown in Figure 4.7.

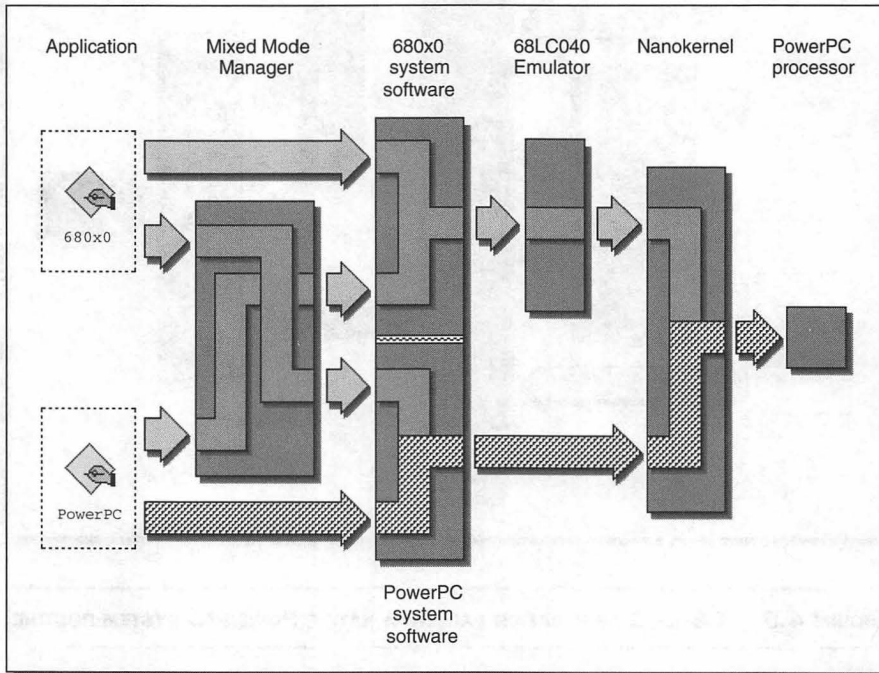


FIGURE 4.6 A POWERPC APPLICATION CALLING A NATIVE POWERPC SYSTEM ROUTINE.

While the previous figures show what appears to be a complex handling of Toolbox calls, your role as a programmer is—fortunately—minimal. The system does most of the work for you. This is evident in the running of an unmodified 680x0 application. Even without any porting or recoding, most existing 680x0 programs run fine on a Power Macintosh. When the time comes for you to port existing 680x0 code to PowerPC code or to write a new native PowerPC program, your involvement increases. But only minimally. Mode switching situations that require your input will be discussed later.

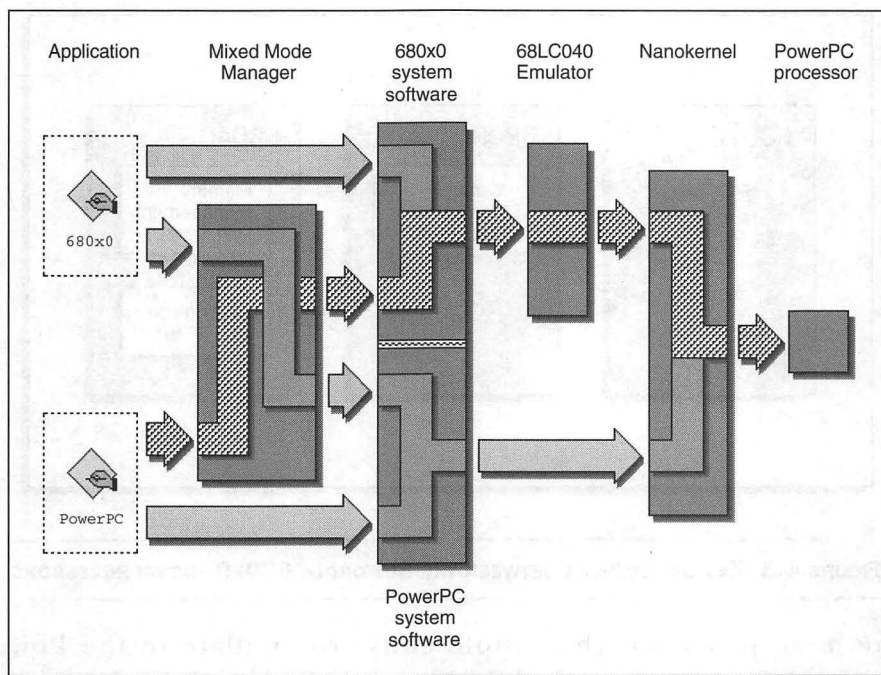


FIGURE 4.7 A POWERPC APPLICATION CALLING A 680x0 SYSTEM ROUTINE.



Today's quiz: If an existing 680x0 application seems to run fine on a Power Mac, why port it over to native PowerPC code? To make it run even better! A 680x0 application is not optimized for PowerPC performance, and is spending more time in the 68LC040 Emulator than a native PowerPC application. Any time a program enters the Emulator, it is running slower than if it were running directly on the PowerPC chip.

THE 68LC040 EMULATOR

All late-model Macintosh computers—aside from the Power Macs—contain either the 68020, 68030, or 68040 microprocessor. Figure 4.8 shows the essential differences between these 680x0 series chips.

Programming the PowerPC

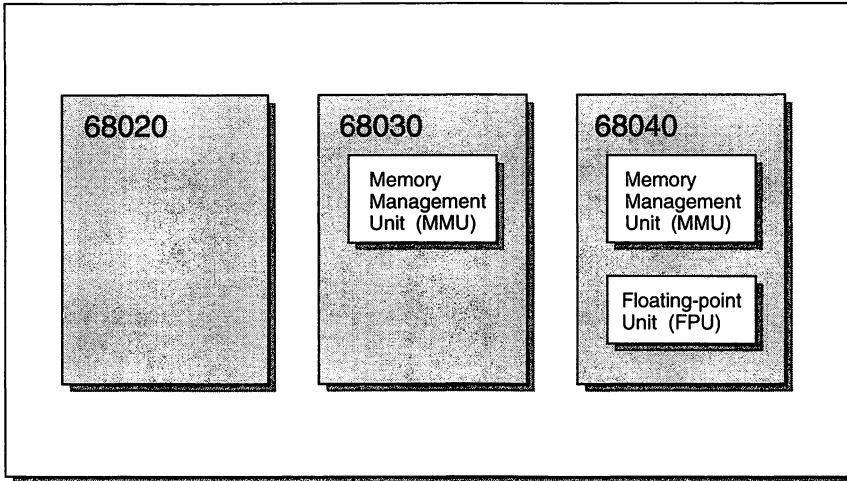


FIGURE 4.8 KEY DIFFERENCES BETWEEN THE MOTOROLA 680x0 MICROPROCESSORS

The microprocessor that Apple chose to emulate in the Power Macintoshes is the Motorola 68LC040—a derivation of the 68040. The 68LC040 microprocessor is a 68040 microprocessor without the floating-point unit (FPU). The emulated version of the 68LC040 goes a step further and removes support for the memory management unit (MMU). While foregoing emulation of the MMU and FPU may seem like a step backwards, Apple did so for good reason.

Whether an instruction reaches the PowerPC chip via the emulator or directly from, say the Toolbox, all memory management is handled by the MMU built into the PowerPC. Thus, there is no need for the 68LC040 Emulator to include MMU emulation.

As in the case of the memory management unit, there is no need for the 68LC040 Emulator to emulate floating-point instructions—the PowerPC has its own floating-point unit.



Since it doesn't include a floating-point instruction set, the 68LC040 Emulator can't handle an application that relies on a floating-point coprocessor being present. This should rarely be an issue. Since many Macs don't come equipped

with a floating-point unit, most 680x0 software packages don't make the assumption that one is present.

When considering instruction sets, a 68040 without the floating-point unit or the memory management unit more closely resembles that of a 68020. In fact, don't be surprised when a call to the `Gestalt()` function tells you just that. If an application passes `Gestalt()` a selector of `gestaltProcessorType` while running in emulation mode on a Power Mac, `Gestalt()` will return a value of `gestalt68020`. Here's that call:

```
OSErr  err;
long   response;

err = Gestalt( gestaltProcessorType, &response );

// response will equal gestalt68020 after the call
```



The Macintosh Centris 610 is a 68LC040 processor-based Mac. So the hardware of the Centris 610 is what the 68LC040 Emulator software is patterned after. If an application executes properly on a Macintosh Centris 610 it should run properly on the 68LC040 Emulator of a Power Macintosh.

THE MIXED MODE MANAGER

Software, whether designed for a 680x0-based Mac or a PowerPC-based model, run in a mixed environment of 680x0 system software and PowerPC system software on a Power Mac. The ROM-based emulation software—the 68LC040 Emulator—and the functions that comprise the Mixed Mode Manager allow applications to run in this mixed environment.

Instruction Set Architecture

Every type of microprocessor has an *instruction set architecture*—a group, or set, of instructions that it recognizes and works with. The 68LC040 Emulator, which mimics a Motorola 68LC040 microprocessor, has its own instruction set architecture. The PowerPC microprocessor has its own instruction set architecture—one that differs from that of the 68LC040 Emulator.

When a single application running on a Power Macintosh makes calls to both 680x0 system software routines and PowerPC system software routines—that is, to both nonported and ported functions—*mode switches* occur. A mode switch means that the Power Mac goes from executing code in one instruction set to executing code in the other instruction set. When this happens, the Mixed Mode Manager is responsible for overseeing this mode switch.

Until the system software has been completely ported to native PowerPC code and all 680x0 applications are ported and recompiled to native PowerPC applications, a mechanism such as the Mixed Mode Manager is necessary.

Cross-Mode Calls

When a 680x0 application makes a call to a PowerPC routine, the call is said to be cross-mode. That is, the mode switches from that of the 68LC040 Emulator to native PowerPC mode. The same applies when a PowerPC application calls a 680x0 routine. Figure 4.9 shows both types of cross-mode calls.

Apple's intentions are for cross-mode calls to be handled transparently by the Mixed Mode Manager. No intervention is necessary on the part of an application's user, and little is necessary on the part of the programmer.

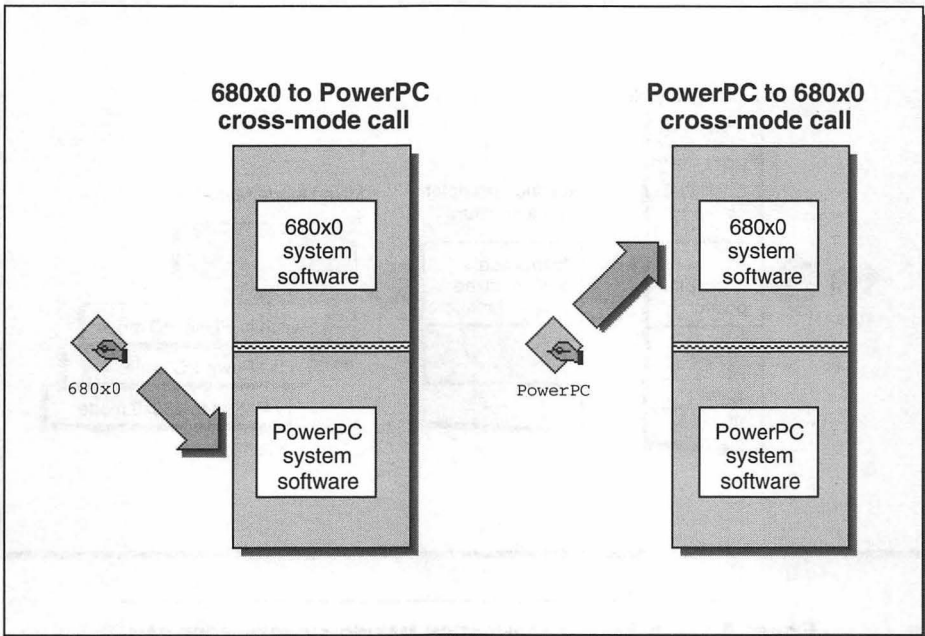


FIGURE 4.9 A 680x0 APPLICATION AND A POWERPC APPLICATION MAKING CROSS-MODE CALLS.

680x0 to PowerPC Cross-Mode Calls

When a 680x0 application calls a native PowerPC routine such as a Memory Manager or QuickDraw function, the emulator is called—but not directly. First, the Trap Manager checks the trap dispatch table. If the routine is a native PowerPC function, then the table holds a pointer to a *routine descriptor*. A routine descriptor is a data structure that holds information about a routine, such as the parameters expected for a call to the function.

The first field in a routine descriptor is an instruction that invokes the Mixed Mode Manager. Once invoked, the Mixed Mode Manager switches to native PowerPC mode, calls the native PowerPC code, and then returns to the 68LC040 Emulator. Figure 4.10 shows a 680x0 to PowerPC cross-mode call.

Programming the PowerPC

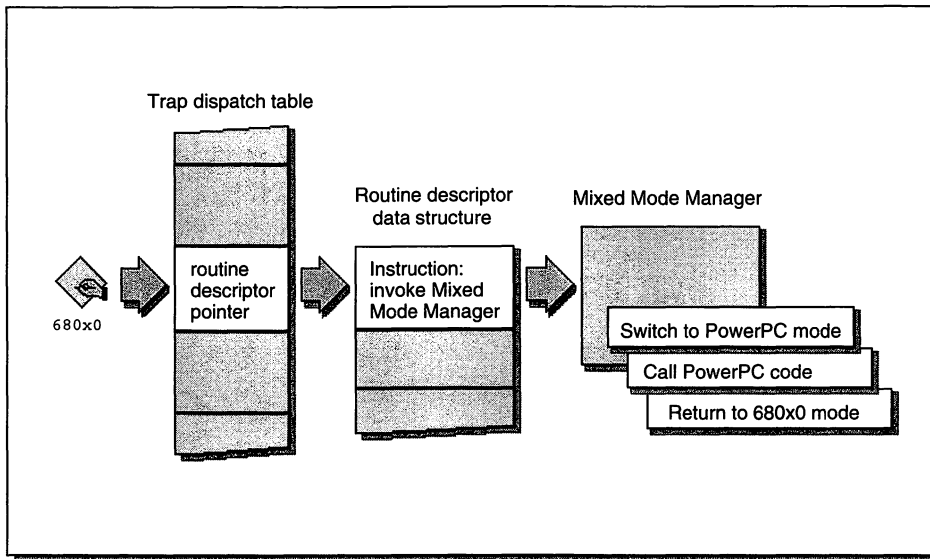


FIGURE 4.10 A 680x0 APPLICATION MAKING A CROSS-MODE CALL.

All 680x0 to PowerPC cross-mode calls are handled implicitly by the Power Mac. The programmer need do nothing to ensure that the call is handled correctly.

PowerPC to 680x0 Cross-Mode Calls

On occasion, a PowerPC application will find it necessary to call a system routine that hasn't been ported to native code. When that happens, the operating system invokes some Toolbox interface glue code. This glue first checks the trap dispatch table to get a pointer to the code that makes up the 680x0 routine. The glue then passes this information on to the Mixed Mode Manager. The Mixed Mode Manager then switches to the 68LC040 Emulator mode and calls the 680x0 routine. When the 680x0 routine has executed, the Mixed Mode Manager switches back to native PowerPC mode. Figure 4.11 shows a PowerPC to 680x0 cross-mode call.



Some apparent system software routines don't, in fact, exist in the System file or the ROM chips of your Mac. Instead, they are implemented by your development system. Such a call is a glue routine. In response to a call to a glue routine, your development system executes some combination of existing system software routines or some assembly language instructions to carry out the task or tasks of the glue routine.

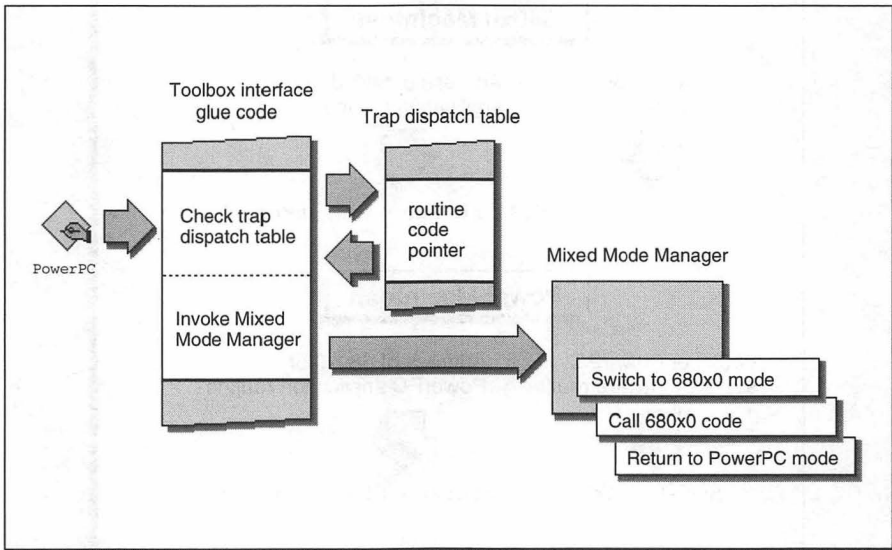


FIGURE 4.11 A POWERPC APPLICATION MAKING A CROSS-MODE CALL.

The Programmer's Role in Mode-Switching

You've seen that 680x0 applications that call PowerPC code need no help from the programmer. The same is true with most PowerPC application cross-mode calls to 680x0 routines. There are, however, occasions when you, the PowerPC programmer, will be responsible for aiding the Mixed Mode Manager in its tasks.

Programming the PowerPC

Certain system software routines accept a pointer to another routine as one parameter. As the system routine executes, it makes use of the function whose address was passed to it. Because the function whose address was passed could be written in 680x0 code or native PowerPC code, a situation such as this requires special consideration on the part of the programmer. Figure 4.12 illustrates why.

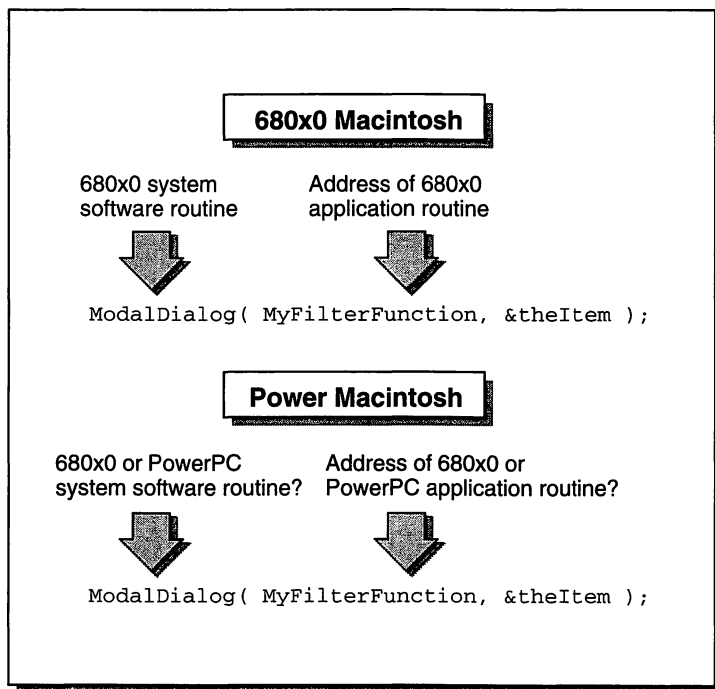


FIGURE 4.12 A PROCEDURE POINTER CAN BE THE ADDRESS OF EITHER A 680x0 ROUTINE OR A POWERPC ROUTINE

On a Power Macintosh, you won't know exactly which system routines have been ported to native PowerPC code. Additionally, the system routine won't know if the code you wrote for the passed function was written in older 680x0 code or optimized, native PowerPC code. For a function that makes use of a `ProcPtr`—a procedure pointer—you'll have to use a routine descriptor. The routine descriptor tells the system routine the

instruction set architecture—680x0 or native PowerPC—of the code whose address is being passed to it. The descriptor also gives the system software information about the parameters that the ProcPtr routine uses.



A working example of the use of routine descriptors is presented in the sample program listing in Chapter 7.

ProcPtr is the data type of a generic procedure pointer. There are also other procedure pointer types that are based on this type. Any system routine that requires any type of procedure pointer as a parameter will require a routine descriptor. When the routine descriptor is then used in place of the procedure pointer parameter, the called system routine will be able to properly invoke the Mixed Mode Manager.

CHAPTER SUMMARY

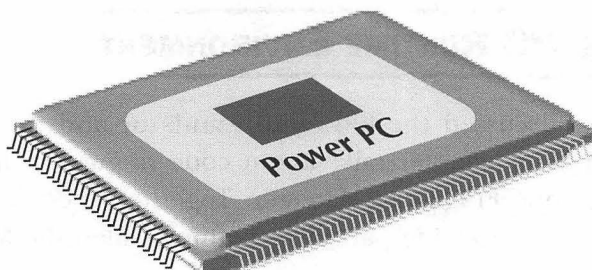
Code runs quicker on a PowerPC-based Mac than it does on most 680x0-based machines. And further speed gains come about when code is recompiled using a compiler designed to optimize code for the PowerPC. With the introduction of the first Power Macs came an updated Toolbox with about 10 percent of the Toolbox routines rewritten in C and recompiled into fast native PowerPC code. While 10 percent may seem like a low figure, Apple has determined that 90 percent of the calls to Toolbox functions are made to just 10 percent of the Toolbox routines—the 10 percent that have been ported.

To avoid the problems of having a mix of 680x0 routines and native PowerPC routines in the Toolbox, Apple has made two important additions to the Macintosh system software—the 68LC040 Emulator and Mixed Mode Manager. The Power Macs contain new ROM chips and a new System file that together contain these two additions that support software running in a mixed environment.

Any program that makes a call to a 680x0 system software routine will temporarily jump into the 68LC040 Emulator of the Power Mac. The

Programming the PowerPC

Emulator is a software program that resides in the ROM chips of each Power Mac. Its purpose is to convert 680x0 instructions to PowerPC instructions, and then pass these converted commands to the PowerPC microprocessor for execution.



CHAPTER 5

POWERPC SYSTEM SOFTWARE CODE FRAGMENTS

The Macintosh system software has undergone many changes in order to support the Power Macs and the PowerPC chip. Of these changes, the most notable is the support of *fragments*. While executable code such as applications, system extensions, and code resources still go by those same names to users of all Macintosh models, to developers of PowerPC software they are all called *code fragments*. That lends a commonality to all types of executable code—which is of benefit to both the developers that write code and the system software responsible for executing it.

This chapter takes a thorough look at code fragments—including the import library. This type of fragment allows developers of multiple applications to eliminate redundant code from their programs. Also covered in this chapter is the new system software that supports fragments—the Code Fragment Manager.

THE POWERPC RUNTIME ENVIRONMENT

Chapter 4 discussed the 68LC040 Emulator and the Mixed Mode Manager. This chapter talks about code fragments, and their handling by the Code Fragment Manager. Together, these system software components make up a big part of something called the Macintosh *runtime environment*.

What the Runtime Environment Is

The *runtime environment*, or *runtime architecture*, of a computer is the combination of executable code and system software that runs it. You—through your programming skills and your programming development system—are responsible for creating the executable code. The second part of the runtime environment—the Macintosh system software—is then responsible for managing the details of loading, managing, and executing the applications you’ve developed.

Each computer has its own runtime environment. While still a Macintosh, the PowerPC processor-based Macintoshes have a runtime environment that differs in many ways from that of the 680x0 processor-based Macs.

A New Runtime Environment— And Why It Was Needed

While the new Power Macs are capable of running both old and new applications, great speed increases are only noticed when a new PowerPC program is executing. That’s because only the native PowerPC applications will be able to take advantage of all of the extensive changes that are in the new PowerPC runtime architecture.

Even without the arrival of the RISC processor, the Macintosh runtime architecture was due for an overhaul. Over the years, Apple has improved the environment in bits and pieces. But it has always been

based on the same runtime environment that was designed for the original Macs—and the limitations of those first computers.

The first Macs had just 128 K of RAM—one-eighth of 1 Megabyte. The runtime environment was designed with that in mind. As such, limitations were imposed upon the designers of the Mac environment. Now, with inexpensive and readily available RAM, those limitations no longer exist. Yet they still stifle the runtime environment of the entire line of 680x0-based Macintosh computers.

A second limiting factor in the advancement of the Mac runtime architecture also involves memory. The original Macs did not come with a hard disk or memory management unit. That, along with extremely limited RAM, made program segmentation a necessity. Because an entire application could not be placed into memory at one time, programs had to be divided into segments that would be loaded and unloaded during the course of program execution. Abundant RAM and virtual memory have now eliminated the need for segmentation—though the 680x0 family of Macintoshes still require it.

Apple had to redesign the Macintosh hardware architecture to accommodate the PowerPC chip. At the same time, they had to also make changes to the runtime architecture to support the new processor chip. With a large amount of time and money already invested, and many system software changes required, what better time to revamp the entire runtime architecture? That's how Apple felt, and that's what they did.

IMPORT LIBRARIES

An *import library*—also called a *shared library* or a *dynamically linked library*—is a collection of compiled functions that can be used by one or more applications. The code of an import library may be stored in ROM or in a resource, but more typically it is held in a library file.

Linked Libraries and Import Libraries

The idea of a library of compiled code is not new to the Macintosh—both the 680x0 Macs and the Power Macs make use of *linked libraries*. A linked library is a file of compiled code that gets added to an application when the application is built—that is, at link time. The MacTraps library that programmers add to a project file is an example of a linked library.

MacTraps, and other libraries such as MacTraps2 and Graf3D, are supplied to you along with the rest of the development environment you purchase. But programmers can also make their own linked libraries. The first step to creating a linked library is to write the source code for one or more functions. How that source code becomes a library depends on the development environment used. If you were using a Symantec compiler, you'd then create a new project, add the file that contained the functions to the project, then select **Build Library** from the Project menu. The result would be a library that could be added to any other project. Once added to a new project, functions in the library could be called by source code in the new project. When it comes time to build an application, the linker combines object code, resources, and the code for any library functions that are called.

Let's examine an example. Assume that source code in a file named Test.c calls one function in a library named MyLib. I'll call the compiled Test.c code Test.o. When it comes time to build an application, the linker will combine the Test.c object code with the resources and with the library code for the one called function. The result is shown in Figure 5.1. Note that the final application contains code from the library.

While the idea of a library of compiled code isn't new to the Macintosh, import libraries are. With import libraries and the PowerPC, the situation changes from that pictured in Figure 5.1. Like a 680x0 linked library, an import library also can contain the object code of user-written functions. But when it comes time to build an application, the library code does not end up in the final application. Instead, only a reference to the code in the library makes its way into the application. When a user launches the application, both the application and the import library code are loaded into memory. This means that in order

for the application to run, the import library must be present. If the application is distributed to others, the import library must be distributed as well. Figure 5.2 shows this.

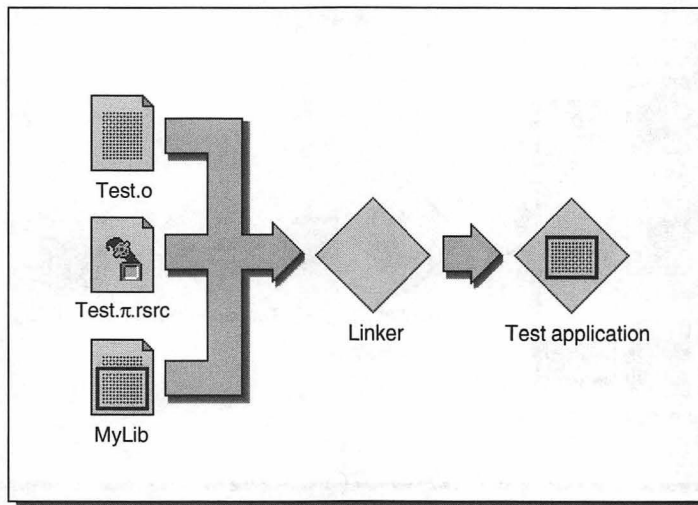


FIGURE 5.1 USING A LINKED LIBRARY AS PART OF AN APPLICATION.



NOTE

You may have noticed that I said that an application holds a *reference* to import library code, rather than a *pointer* to library code. The application doesn't actually hold a specific address—that's not known until the application is launched and the application and the import library are loaded into memory. Rather, the application keeps a placeholder, or marker, that will be filled with the library code address at application startup. More is said about this in the Code Fragment Manager section of this chapter.

Advantages of Import Libraries

At first glance, the fact that an import library must accompany an application may seem like import libraries are actually a regression from the

Programming the PowerPC

older linked library method. Closer examination, however, shows that import libraries have some important benefits over linked libraries.

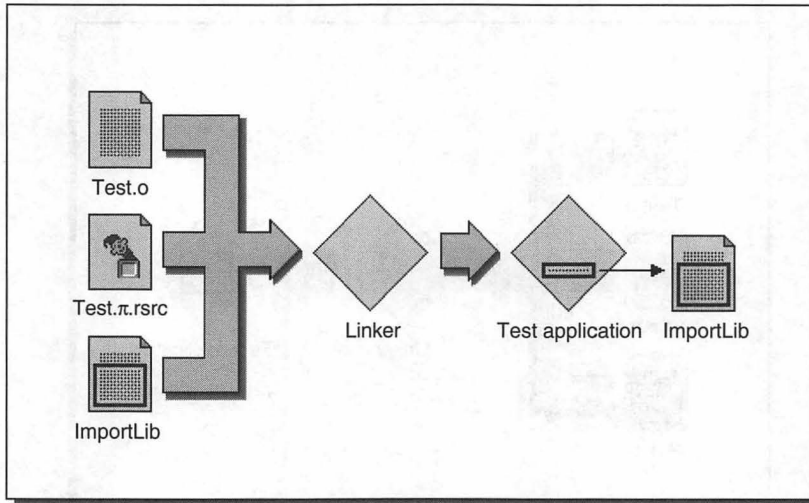
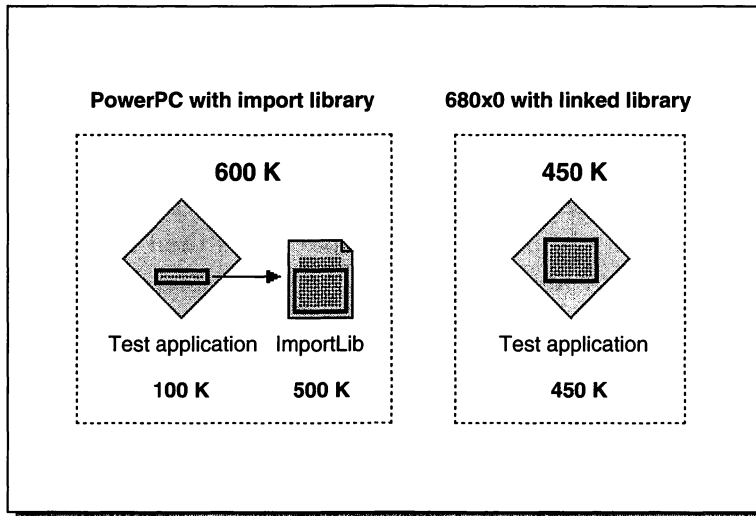


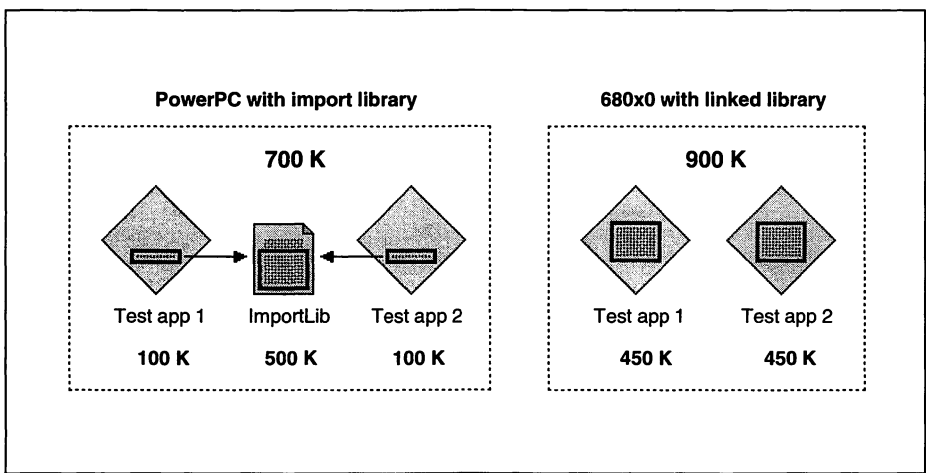
FIGURE 5.2 USING AN IMPORT LIBRARY AS PART OF AN APPLICATION.

An application that uses an import library is smaller than one that doesn't. If only one application uses the import library, the advantage of this decreased size will be lost. That's because the combined size of the application and the import library will occupy as much or more disk space as an application created with linked libraries—as shown in Figure 5.3. The PowerPC test application in this figure includes a reference to some of the code in an import library.

The advantage of using import libraries comes when more than one application uses the same import library. Redundant code from the different applications can be placed in a common import library where both applications can make use of it. This is shown in Figure 5.4, where the two PowerPC applications both contain references to the same import library code.



**FIGURE 5.3 THERE IS NO DISK SPACE SAVINGS
WHEN ONE APPLICATION USES AN IMPORT LIBRARY.**



**FIGURE 5.4 THERE IS A DISK SPACE SAVINGS
WHEN MULTIPLE APPLICATIONS USE AN IMPORT LIBRARY.**

Programming the PowerPC

What is the likelihood of two applications sharing a large amount of common code? Actually, quite high—considering the amount of interface-related code that is redundant from one Mac application to another.

Centralizing the code that is common to multiple applications has advantages beyond the saving of disk space. When two applications that share a common import library are both loaded into RAM, only one copy of the import library is loaded. Both applications then make use of the same import library code, as shown on the left side of Figure 5.5. On a 680x0 processor-based Mac, two applications will occupy more RAM than the two on the PowerPC processor-based Mac—even if both 680x0 applications were linked with the same linked library. That's shown on the right side of Figure 5.5.

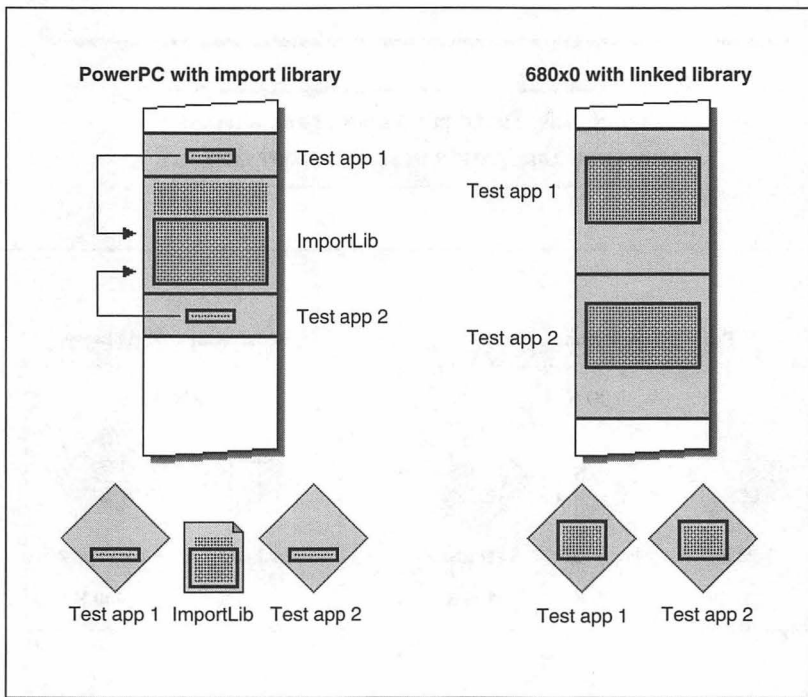


FIGURE 5.5 MULTIPLE APPLICATIONS USING AN IMPORT LIBRARY RESULT IN REDUCED RAM USAGE.

Memory considerations aren't the only reasons for opting to use import libraries. Obviously, eliminating the need to duplicate code means less code has to be written. It also allows for easier upgrading and bug fixes to applications. An enhancement that is added to a single import library will benefit all applications that make use of that library.



Chapter 10 walks through the development of a small import library and two simple applications that make use of it.

In the PowerPC runtime environment, executable code exists as a fragment. An import library is a fragment that exports its functions and global variables for use by other fragments. When the Code Fragment Manager loads an application fragment, it also automatically loads the code of any import libraries that the application fragment uses—unless it was previously loaded and is still in memory.

CODE FRAGMENTS

On the Power Macintosh, any group of executable code—whether it be an application, extension, or code resource—is considered to be a code fragment. The remainder of this chapter looks at code fragments and how they are handled by the Code Fragment Manager.

About Code Fragments

The PowerPC runtime environment varies from that of the 680x0 Macs in the way it contains and handles executable code. On a Power Mac, any one group of executable code—and the data that accompanies it—is considered a *code fragment*. All units of executable code bear the same title—code fragment. Fragments can, however, be thought of in terms of their different purposes.

An *application* code fragment is just that—an application. Like a 680x0 application, a PowerPC application requires no other code, aside

Programming the PowerPC

from system software, to execute. Optionally it can, however, make use of the code in other fragments.

An *import library* code fragment holds code and data accessed by one or more other fragments. Unlike an application fragment, an import library requires at least one other fragment to call the routines it holds.

A *code resource* code fragment holds executable resource code. A menu definition, or MDEF, is one example.

Extension code fragments add to the capabilities of other fragments. QuickTime is an example of an extension fragment.

While there are different types of code fragments, all fragments use the same techniques to hold code and data, and to access the code and data of other fragments. This common structure makes it easy for the system software to work with executable code. This new structuring of code also means that new system software was necessary. The new system software routines that manage fragments is collectively called the Code Fragment Manager.

The Code Fragment Manager

While all code fragments exist as independent units of executable code, code fragments are not able to run on their own. For example, an application fragment makes use of system software, which is itself a fragment. An import library fragment contains the executable code for functions, but doesn't have a main function—it runs in conjunction with the application fragment that invokes it. The Code Fragment Manager, or CFM, is responsible for coordinating the interaction between the different fragments that, together, make up a single application.

When an application is built, the linker establishes which fragments hold the code that the application requires. When the final application is launched, the Code Fragment Manager loads the necessary fragments into memory and supplies each individual fragment with memory addresses of the other related fragments. As the application runs, each fragment can, under the control of the Code Fragment Manager, both export and import information to and from the other fragments.

When a program's user launches an application, the code fragment or fragments that the application consists of are loaded into memory by the Code Fragment Manager. The Code Fragment Manager relies on a system of symbols to prepare each segment that it loads. A fragment that makes use of code external to itself will have a series of symbols that denote which routines are called—but not contained in—the fragment. The import library that contains the code for the called routines will have symbols matching those in the calling fragment. From these symbols the Code Fragment Manager creates the pointers necessary to relate fragments to one another.

When the Code Fragment Manager has resolved all the fragment relationships, it loads the fragments into memory. If an application uses an import library fragment that in turn uses code from still another import library, the last import library will be loaded first. With memory addresses established for this third fragment, the Code Fragment Manager can then load the second fragment, and then finally the first fragment. Figure 5.6 illustrates this.



When a fragment needs to be loaded, the operating system invokes the routines that make up the Code Fragment Manager. The operation of the Code Fragment Manager is transparent to the user, and to the programmer. Applications you write will rarely, if ever, have to explicitly make calls to the Code Fragment Manager.

A code fragment contains both executable code—such as compiled functions, and data—such as global variables. The Code Fragment Manager loads the code and the data of a fragment into separate *sections*, or regions, of memory. These two sections do not have to be in—and generally aren't in—contiguous areas of memory.

Once the code and data sections of a fragment are loaded into memory, they will not be moved. Because a fragment contains numerous pointers to information located in other fragments, the moving of one fragment would involve significant updating of the pointers in one or more other fragments. The Code Fragment Manager resolves all the frag-

Programming the PowerPC

ment pointer information one time—when it loads a fragment. By locking a fragment in memory, the Code Fragment Manager will not have to go through this effort again.

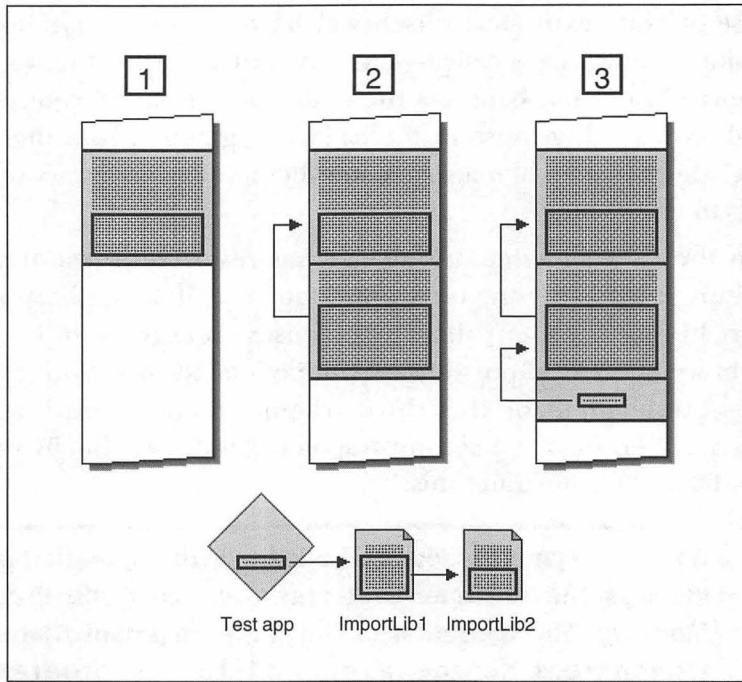


FIGURE 5.6 THE CODE FRAGMENT MANAGER LOADS CODE INTO MEMORY.



Programmers of the 680x0 family are used to segmenting their programs. On a PowerPC, this is no longer necessary. The code section of a fragment is not segmented. The responsibility for juggling code in memory no longer belongs to the programmer or to the application. Old habits can be hard to break—so if a programmer inadvertently leaves any segmentation directives in a source code file, a compiler designed for the PowerPC will simply ignore them. And once a PowerPC application is running, the system soft-

ware will ignore any `UnloadSeg` calls that may have been included by the programmer.

In a 680x0 application, the maximum size of an application's global variables is 32 K. That barrier is also removed in PowerPC applications—there is no segmentation, and no size limit imposed on a fragment's data section.

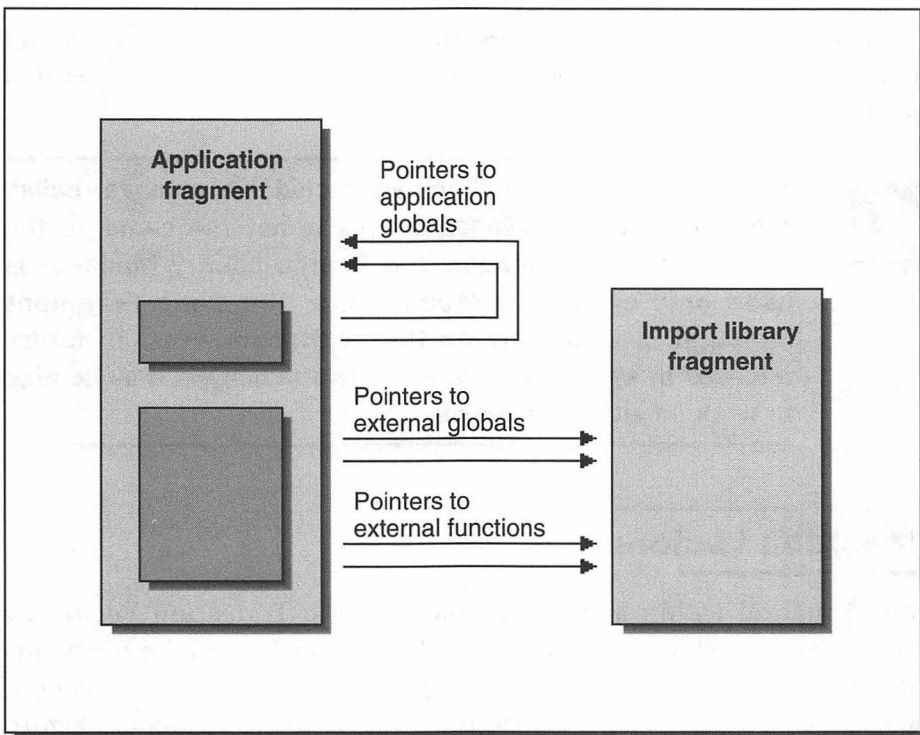


FIGURE 5.7 A FRAGMENT USES POINTERS TO KEEP TRACK OF GLOBAL VARIABLES AND EXTERNAL FUNCTIONS.

If the code and the data of a fragment were loaded in contiguous memory, the fragment's code would always know the location of the fragment's data. The data, such as global variables, could always be at some prede-

Programming the PowerPC

finned offset from the code—the compiled functions that use the data. Since the code section and the data section of a fragment are not required to be in contiguous memory, the code must be provided with the locations of data in the data section. In a similar fashion, a fragment that makes use of global variables and functions that are in a different fragment must also be provided with references to where this external code is located in memory. In all cases, pointers give a fragment the information it needs—as shown in Figure 5.7.

To keep track of these pointers, the Code Fragment Manager relies on a fragment's Transition Vectors and its Table of Contents—topics that are about to be covered.



The 680x0 runtime environment includes a manager called the Shared Library Manager. This is not the same as the Code Fragment Manager. The Shared Library Manager is used only on 680x0 Macs, while the Code Fragment Manager is used only on Power Macintoshes. In future releases of system software the two managers may be able to work on either processor.

Transition Vectors

Every function within a fragment has a single Transition Vector, or TVector, associated with it. The function code is located in the fragments code section, and the TVector is located in the fragment's data section. Each TVector serves as a pointer to the code of a single function. Figure 5.8 shows a fragment with two functions. Note that the code section and data section of the fragment have been loaded in noncontiguous parts of memory—as is usually the case.

TVectors exist not for the benefit of the fragment they appear in. Instead, they are used as an aid to other fragments. Using Figure 5.9 as a reference, let's examine an example. Imagine that code in one fragment calls a function whose code is in a different fragment. In Figure 5.9, the fragment on the left can call either FunctionA or FunctionB—both of

which are located in the fragment pictured to the right. In either case, the calling fragment first goes to the address of a TVector in the called fragment. That TVector then leads to the code that makes up the function.

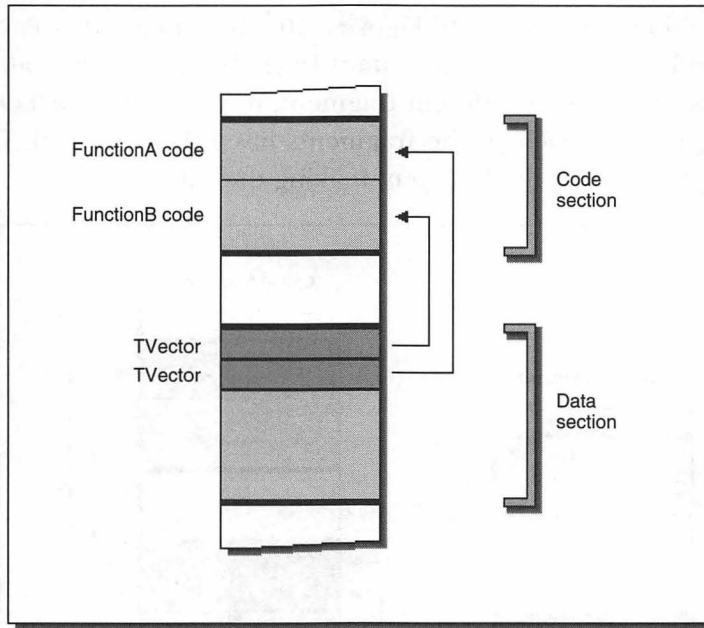


FIGURE 5.8 A FRAGMENT'S TVECTORS POINT TO THE FRAGMENT'S EXECUTABLE CODE.

In Figure 5.9, the pointers to the two external TVectors are shown grouped together. In fact, this is how they would appear in memory. Grouped along with the addresses of the TVectors would be other pointers that the fragment would use to keep tabs on information it needs. Collectively, this group of pointers is called the fragment's Table of Contents.

The Table of Contents

When a PowerPC application is built, the linker creates a Table of Contents, or TOC, for each fragment that contains code used by the application. The linker places a symbol in the Table of Contents of a frag-

Programming the PowerPC

ment for each external function that is used by the fragment. When the completed application is launched, the Code Fragment Manager replaces the symbols with the addresses of the TVectors that lead to the code of each function. Figure 5.10 updates Figure 5.9 by placing emphasis on the Table of Contents. In Figure 5.10 it is assumed that FunctionC makes a call to FunctionA and FunctionB. If code in the called fragment accesses code in a different fragment, it too will have a TOC. Note that in Figure 5.10 each of the fragments has a TOC, though I've only shown the TVectors in the fragment making the call.

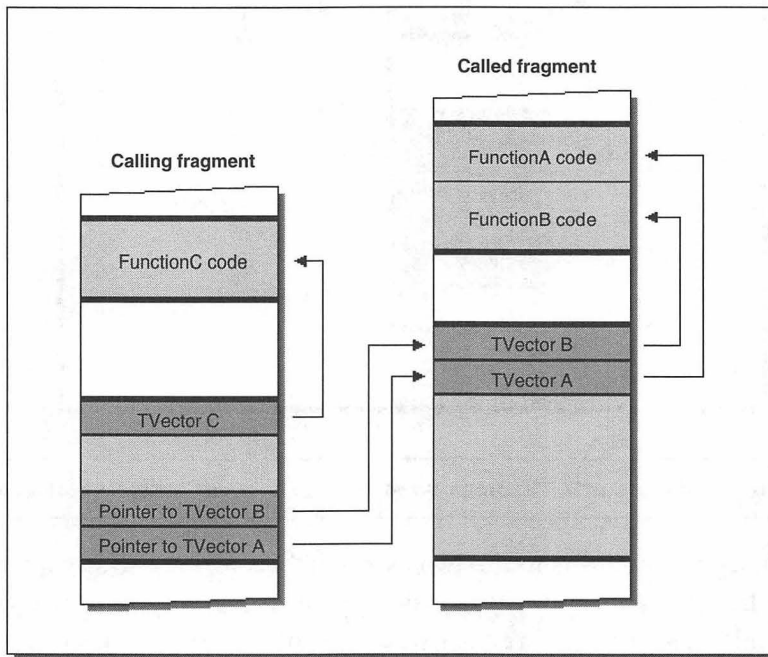


FIGURE 5.9 ONE FRAGMENT ACCESSES CODE IN ANOTHER VIA TVECTORS.



The symbols can't be converted to pointers by the linker—this step must take place at runtime. That's because during the building of the application, the linker doesn't know where in memory the program will be loaded when it is launched.

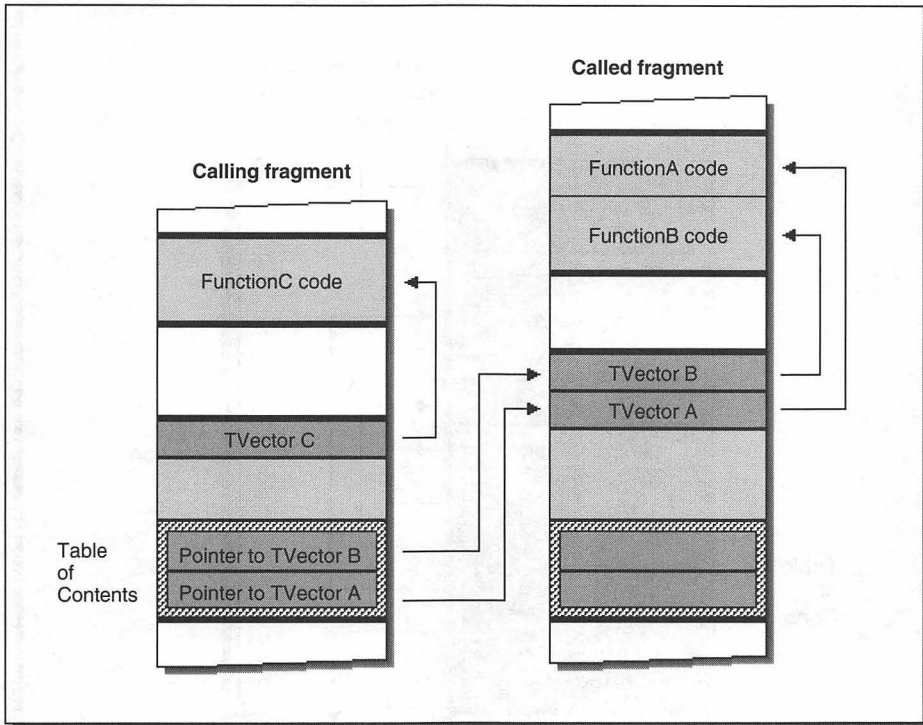


FIGURE 5.10 A FRAGMENT'S TOC HOLDS POINTERS TO THE TVECTORS IN OTHER FRAGMENTS.

A Table of Contents contains more than just pointers to TVectors. It also contains pointers to global variables and static variables. Recall that a static variable is a variable that is local to a function, yet retains its value between function calls. Each global variable used by the fragment—whether the variable appears in the fragment or in another fragment—gets its own Table of Contents entry. Additionally, each group of static variables that appear within the code section of the TOC's fragment get a single entry in the Table of Contents. These pointers allow the code in the code section of a fragment to find and use variables that are scattered about in memory. Figure 5.11 shows how a fragment's data section is structured.

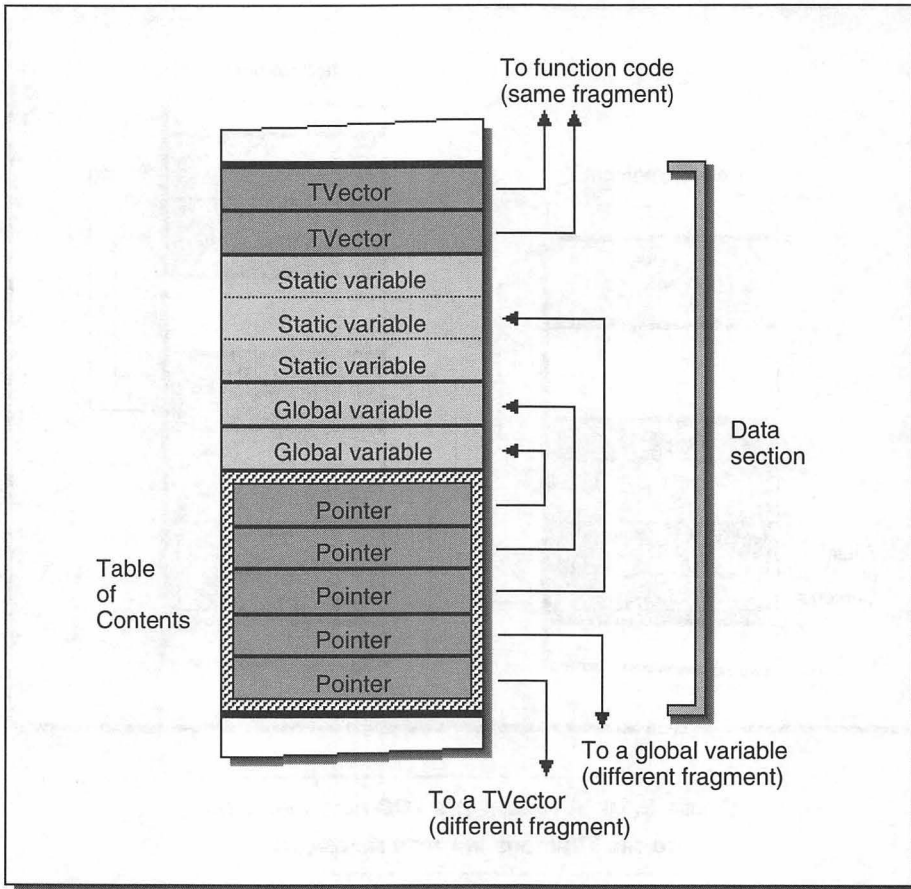


FIGURE 5.11 THE STRUCTURE OF A CODE FRAGMENT'S DATA SECTION.



NOTE

PowerPC applications don't have an A5 world, as 680x0 applications do. Instead, a fragment uses the TOC for the purpose of keeping track of its own data. There is an advantage to the TOC's way of doing things—any fragment can have global data. The A5 world used in 680x0 development allows for global data only in applications.

The following code snippet is for a fragment that declares two global variables, `GrandTotal` and `FinalSales`. The fragment contains two functions—`FunctionA` and `FunctionB`. `FunctionB` contains three static variables—`count`, `index`, and `total`. This fragment makes use of one function, `FunctionC`, that is found in a different fragment. It also uses one global variable found in the second fragment—`MonthlyTotal`.

```
void FunctionC( void );

extern int MonthlyTotal;

int GrandTotal;
int FinalSales;

void FunctionA( void )
{
    // function code here
}

void FunctionB( void )
{
    static int count;
    static int index;
    static int total;
    // function code here
}
```

Figure 5.12 illustrates how memory would look for the first fragment. Notice that the fragment's Table of Contents contains one entry for each global variable—whether the variable is found in the fragment or in a different fragment. It also contains one entry for each group of static variables found in the fragment. A group consists of the static variables found in one function. Finally, the Table of Contents contains an entry for each external function that is used by the fragment.

One final note on TVectors. A function's transition vector actually contains two pointers. The first is the pointer to the code that makes up the function. The second is a pointer to the fragment's table of contents. Thus the second of the two pointers of each transition vectors in a fragment will point to the same address—the address of that fragment's TOC. Why does a function in a fragment need to know where that same

Programming the PowerPC

fragment's TOC lies? So that one function can successfully call a function that resides in a different fragment. Figure 5.13 illustrates this idea.

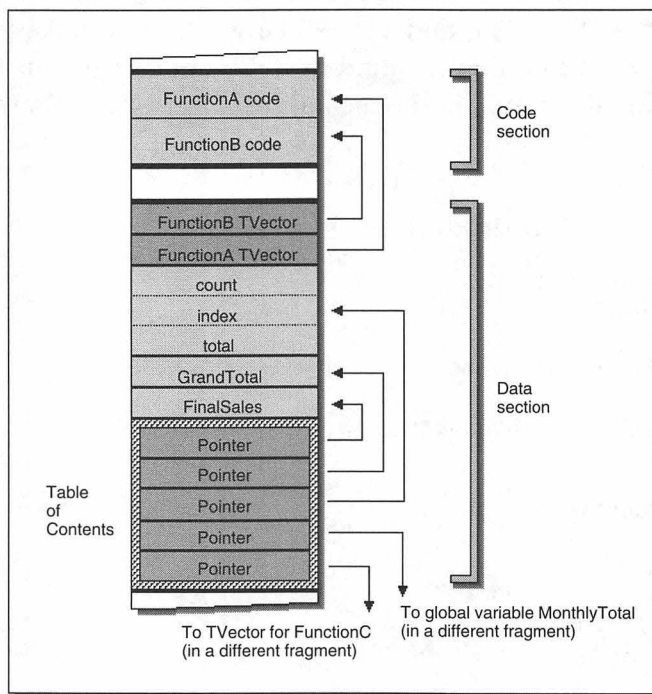


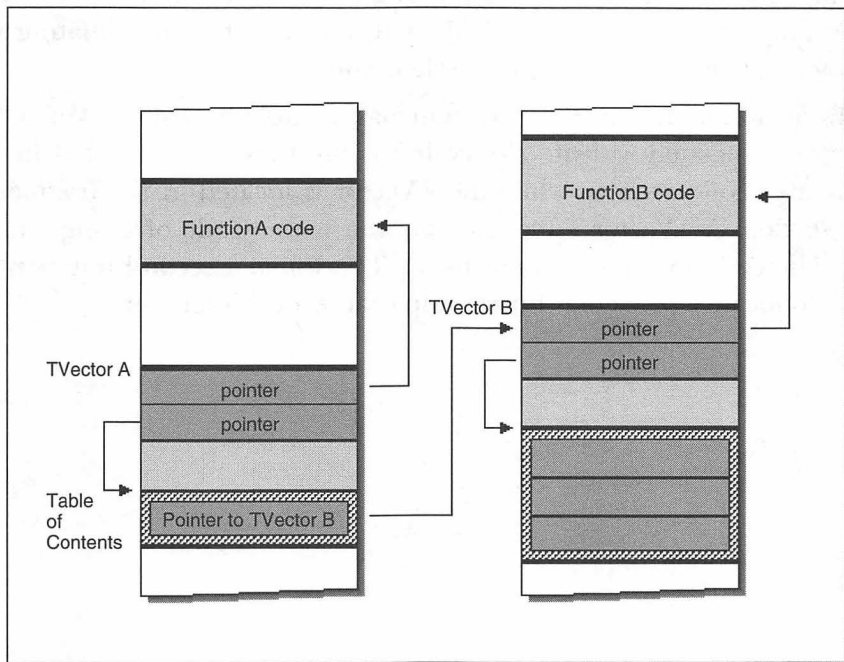
FIGURE 5.12 REPRESENTATION OF A CODE FRAGMENT IN MEMORY.

Figure 5.13 shows that the TVector for function A has a pointer that points to the functions code, and a pointer that points to the fragment's TOC. Should function A make a call to function B, the TVector for function A will be used to determine where the function B code resides.

CHAPTER SUMMARY

The *runtime environment* of the Mac is the combination of executable code and system software that runs it. A programmer, along with his or her programming development system, is responsible for creating the

executable code. The other part of the runtime environment—the Macintosh system software—is responsible for coordinating the details of loading, managing, and running the executable code that the programmer has developed.



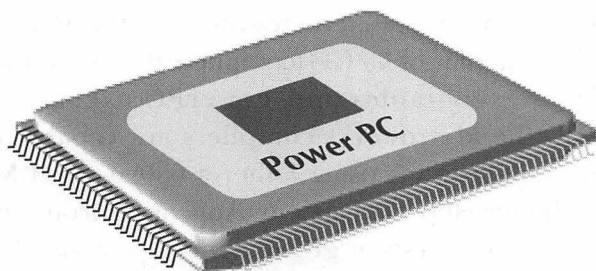
**FIGURE 5.13 A TVECTOR CONTAINS TWO POINTERS—
ONE TO A FUNCTION'S CODE, THE OTHER TO THE FRAGMENT'S TOC.**

For users of the Power Macs, executable code—such as applications, system extensions, and code resources—still go by these same names. For developers of PowerPC software, they are now all called *code fragments*.

An *application* code fragment is a standalone application. Like a 680x0 application, A PowerPC application, like a 680x0 application, requires no other code (aside from system software) to execute. Optionally, however, a PowerPC application can make use of the code in other fragments. An *import library* code fragment holds code and data that is accessed by other

fragments. An import library requires at least one other fragment to call the routines it holds. A *code resource* code fragment holds executable resource code. A control definition, or CDEF, is one example. *Extension* code fragments add to the capabilities of other fragments. QuickTime and Macintosh Drag and Drop are examples of extension fragments. The Code Fragment Manager, or CFM, is responsible for coordinating the interaction between the different code fragments.

Every function within a fragment has a single Transition Vector, or TVector, associated with it. The code for the function is located in the fragment's code section, while the TVector is located in the fragment's data section. A TVector serves as a pointer to the code of a single function. The code in one fragment uses a TVector in a second fragment in order to locate a particular function in that second fragment.



CHAPTER 6

POWERPC COMPILERS

To create a standalone, native PowerPC program for a Power Macintosh you'll need a compiler that was designed to do just that. Older, existing compilers will generate executables that will run on both 680x0-based Macs and PowerPC-based Macs, but the programs won't take advantage of the speed of the PowerPC chip—they won't be native applications.

For years Symantec Corporation has dominated the Macintosh compiler market with their Symantec C++ and THINK C compilers. Version 7.0 was released close to the time that the Power Macintosh computer hit the market. But while the Symantec C++ 7.0 compiler creates programs that can run on the Power Mac, these applications aren't native PowerPC programs. For that you need to get the Symantec Cross Development Toolkit (CDK). This addition to Symantec C++ 7.0 allows you to create native PowerPC applications using either a 680x0-based Mac or a Power Mac as your development system.

Programming the PowerPC

In 1994 Metrowerks released their entry into the Macintosh PowerPC compiler market—a pair of C/C++ compilers that allow the developer to generate 680x0 executables and PowerPC executables. Like the Symantec CDK, the Metrowerks compilers are capable of generating native PowerPC applications using either a 680x0-based Macintosh or a Power Mac as the development system. Appearing from out of nowhere, the Metrowerks compilers are giving Symantec a run for the money. Which compiler will become the most popular remains to be seen. But one thing is for certain—Symantec is no longer uncontested in the Macintosh compiler marketplace.

This chapter covers both the Symantec CDK and the Metrowerks C/C++ compilers in detail—including walkthroughs of the steps it takes to create a standalone PowerPC program using either development system.

THE METROWERKS CODEWARRIOR COMPILERS

For the last several years, programmers who wanted to write C or C++ applications for the Macintosh had very little choice in which compiler they would use. There was Apple's own Macintosh Programmer's Workshop (MPW) or Symantec's C++/THINK C compiler package. Nine out of ten developers selected the Symantec compiler. But now there is a new kid on the block—Metrowerks. Sneak previews of the Metrowerks CodeWarrior C/C++ compilers were available in 1993, and the final versions arrived in 1994. In a very short time, CodeWarrior has become very, very popular.

What Metrowerks Consists Of

Speaking about *the* Metrowerks CodeWarrior compiler isn't entirely accurate. That's because there are actually several CodeWarrior compilers. The first is a Pascal compiler that generates executables that run only on 680x0-based Macs. Obviously enough, this compiler won't be of as

much interest to PowerPC programmers as compilers that create PowerPC code. Thus, the Pascal compiler won't be covered in this book. The other Metrowerks compilers are variations of their combined C/C++ compiler. Here's how Metrowerks distributes them:

Bronze Version

A C/C++ compiler that runs only on a 680x0-based Mac and generates executables that will run on 680x0-based Macs. Applications will also run on Power Macs, but they will not be native PowerPC. That is, they won't take advantage of the speed of the PowerPC chip.

Silver Version

A C/C++ compiler that runs on either a 680x0-based Mac or a Power Mac. The applications this compiler generates will be native PowerPC executables that will only run on a PowerPC.

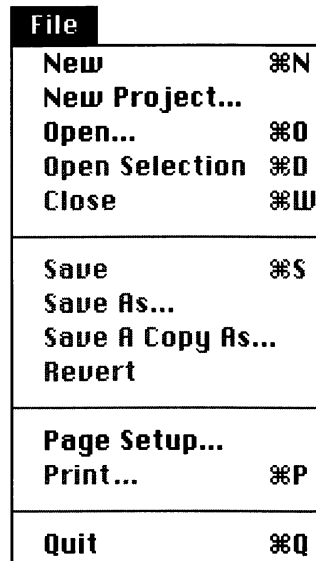
Gold Version

Two separate C/C++ compilers. Both run on either a 680x0-based Mac or a Power Mac. The first compiler, named MW C/C++ 68K, generates executables for 680x0-based Macs. The second compiler, named MW C/C++ PPC, generates native PowerPC applications for PowerPC-based Macs. Using each compiler to generate a separate version of the same program allows the developer to then combine the separate versions into a single fat binary application that will run on a 680x0-based Mac or run native on a Power Mac.

For serious developers, the Gold version is the route to take. If you want to support the millions of Mac owners who have older machines, you'll want your applications to run on 680x0-based Macs. If you also want to support the increasing number of Power Mac owners, you'll want your programs to run on Power Macs. And you'll want your applications to run fast on the Power Macs—that means using native PowerPC instructions. The Gold version of Metrowerks CodeWarrior allows you to create such applications.

Creating a CodeWarrior Project

After following the installation instructions supplied by Metrowerks, double-click on the MW C/C++ PPC icon to launch the PowerPC version of the CodeWarrior C/C++ compiler. This compiler generates PowerPC-only code, but the compiler itself runs on either a 680x0-based Mac or a PowerPC-based Mac. Select **New Project** from the File menu. That menu is shown in Figure 6.1.

A screenshot of the 'File' menu from the Metrowerks CodeWarrior application. The menu is displayed as a vertical list of options. The 'File' title is in a dark box at the top. The menu items are: 'New' with a keyboard shortcut '⌘N', 'New Project...', 'Open...' with '⌘O', 'Open Selection' with '⌘D', 'Close' with '⌘W', a separator line, 'Save' with '⌘S', 'Save As...', 'Save A Copy As...', 'Revert', another separator line, 'Page Setup...', 'Print...' with '⌘P', another separator line, and 'Quit' with '⌘Q'.

File	
New	⌘N
New Project...	
Open...	⌘O
Open Selection	⌘D
Close	⌘W
Save	⌘S
Save As...	
Save A Copy As...	
Revert	
Page Setup...	
Print...	⌘P
Quit	⌘Q

FIGURE 6.1 THE METROWERKS FILE MENU.

A dialog box that allows you to name the new project will open. To keep things organized you'll want to keep the new project in its own folder. First use the pop-up menu in the dialog box to move up a folder—keeping project folders in the main CodeWarrior folder helps the compiler search for files that will be included in your projects. Next, click on the **New Folder** button and type in an appropriate name for the folder. I'll be naming this first Metrowerks program MWdemoPPC, so I've named the folder (6) MW Demo PPC *f*—as shown in Figure 6.2. The “(6)”

refers to the fact that this is a Chapter 6 example. You'll find a copy of this folder on the disk that was included with the book. If you have Metrowerks, you can use the project in the (6) MW Demo PPC *f* folder. I'd suggest you try creating your own version first, however, so that you become familiar with the process.



The fancy letter “f” stands for “folder”, and is created by pressing the letter “f” key while holding down the **Option** key.

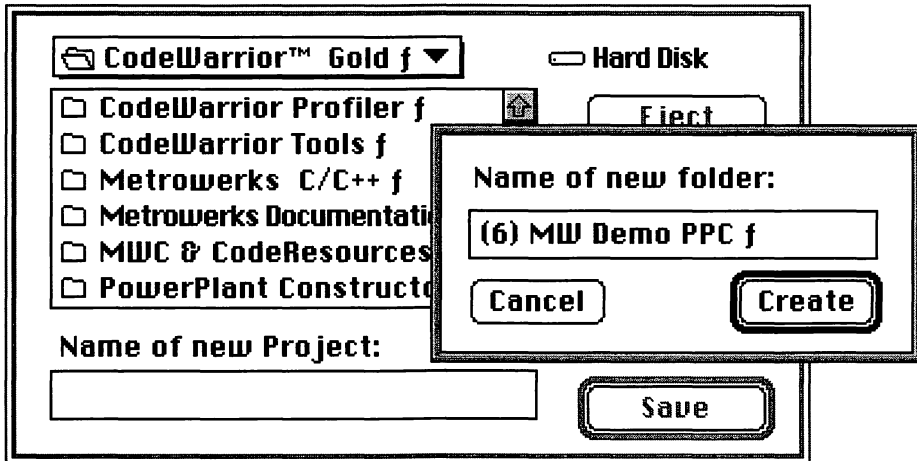


FIGURE 6.2 CREATING A NEW FOLDER TO HOLD THE METROWERKS PROJECT.

Click the **Create** button to create the folder. The New Folder dialog box will close and you'll find yourself inside the new folder. Type in a name for the project—I chose MWdemoPPC.μ—and click the **Save** button. Refer to Figure 6.3.



By convention, CodeWarrior project names end with the extension “μ”. To type this character, press the letter “m” while holding down the **Option** key.

Programming the PowerPC

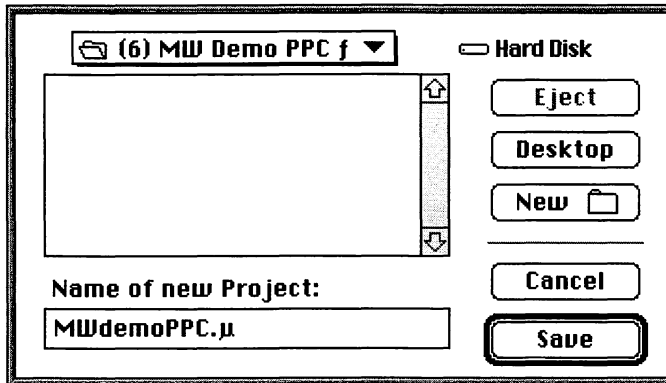


FIGURE 6.3 NAMING THE NEW METROWERKS PROJECT.

After clicking the **Save** button, a new, empty project window will open—as shown in Figure 6.4.

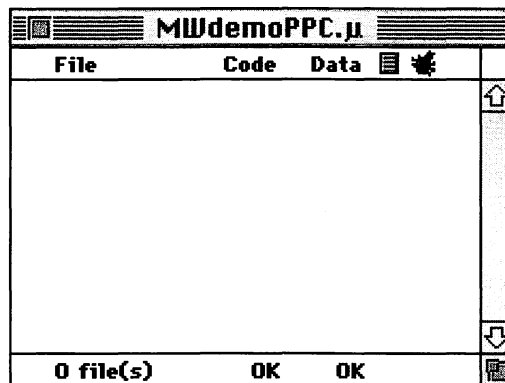


FIGURE 6.4 A NEW, EMPTY METROWERKS PROJECT.

Adding to the Project

To create a new source code file, select **New** from the **File** menu. Select **Save As** from the same menu to give the file a name. I chose the name `MWdemoPPC.c`.

Chapter 6 PowerPC Compilers

Before typing in the source code for the program you'll want to add all the files you'll need for the project. Aside from your source code file there's three Metrowerks libraries you'll want to include in each CodeWarrior project you create: MWCRuntime.Lib, MathLib, and InterfaceLib. These libraries include code necessary for just about every project, so you'll want to play it safe and always add them to a new project. Each of these libraries can be found in folders in the Metrowerks C/C++ f folder. The specific pathnames to each library are listed below. Figure 6.5 graphically illustrates the folder hierarchy.

```
Metrowerks C/C++ f : Runtime f : PPC f : MWCRuntime.Lib
Metrowerks C/C++ f : Libraries f : MacOS PPC f : MathLib
Metrowerks C/C++ f : Libraries f : MacOS PPC f : InterfaceLib
```

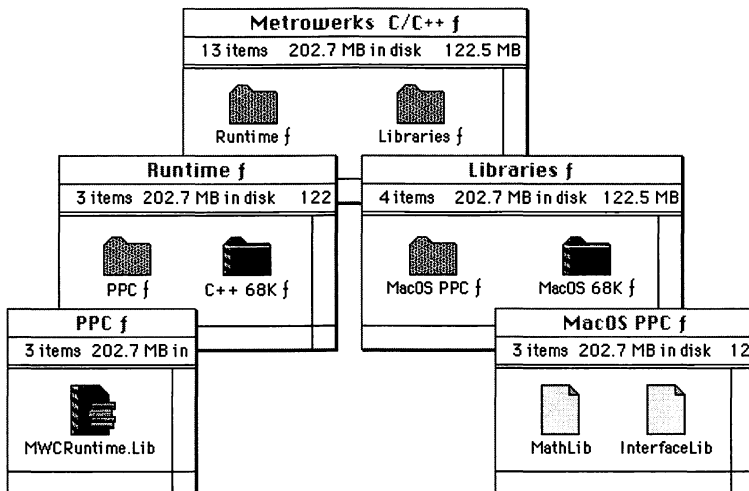


FIGURE 6.5 THE FOLDERS THAT LEAD TO THE THREE LIBRARIES INCLUDED IN ALL METROWERKS POWERPC PROJECTS.



NOTE

The paths listed are for the Metrowerks CodeWarrior release (1.0) as of this book's printing. If a subsequent version of CodeWarrior moves these files, search the folders within the Metrowerks C/C++ f folder for them.

Programming the PowerPC

To add the new source code file and the three libraries, select **Add Files** from the Project menu. This menu is shown in Figure 6.6. To add a library, use the pop-up menu at the top of the dialog box to traverse through the folders. Double-click on a file name to move it to the bottom list in the dialog box. After all four files (MWdemoPPC.c, MWCRuntime.Lib, MathLib, and InterfaceLib) have been added, click the **Done** button. Figure 6.7 shows the Add Files dialog box. The InterfaceLib library has been added—its name is in the bottom list. The MathLib library is about to be added.

Project	
Add Window	
Add File...	
Remove	
Reset File Paths	
Check Syntax	⌘;
Precompile...	
Compile	⌘K
Disassemble	
Remove Binaries	⌘-
Bring Up To Date	⌘U
Make	⌘M
Run	⌘R

FIGURE 6.6 THE METROWERKS PROJECT MENU.

After clicking the **Done** button in the Add Files dialog, you'll see that the four files have been added to the project window. Figure 6.8 shows what your project window should look like at this point.

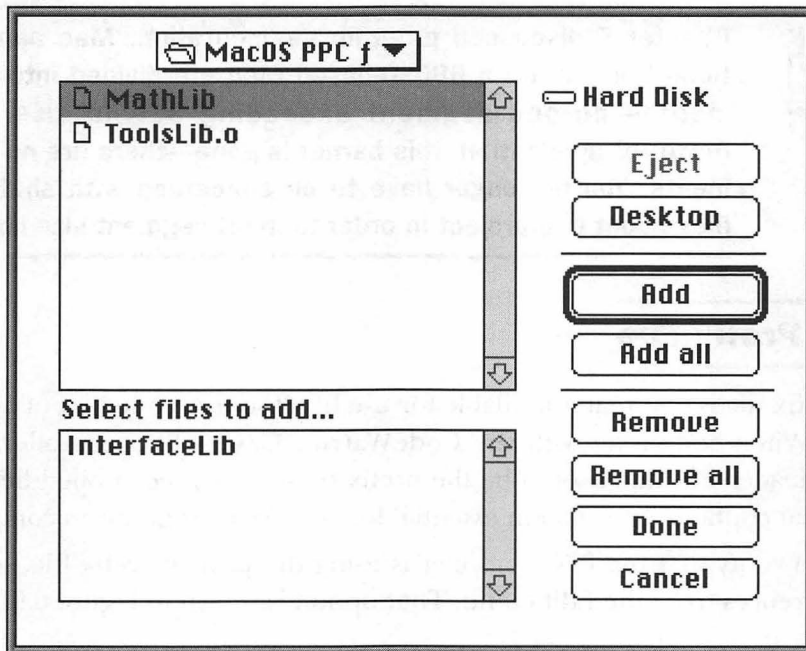


FIGURE 6.7 ADDING FILES TO A METROWERKS PROJECT.

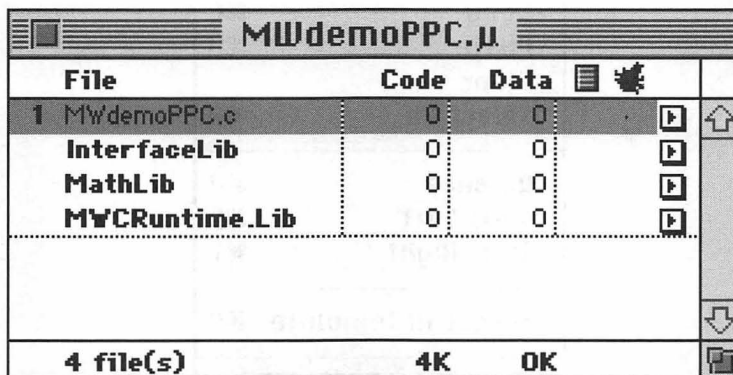


FIGURE 6.8 A METROWERKS PROJECT WITH THE THREE REQUIRED LIBRARIES AND A SOURCE CODE FILE

Programming the PowerPC



Chapter 5 discussed program segmentation. Mac applications that run on a 680x0-based Mac are divided into segments—no one segment exceeding 32K in size. For PowerPC application, this barrier is gone—there are no segments. You no longer have to be concerned with shuffling files about in a project in order to meet segment size limits.

The Prefix File

A prefix file is one that is available for use by all source code files of a project. When compiling with the CodeWarrior C/C++ PPC compiler, the MacHeadersPPC file should be the prefix file—it's a precompiled header file that contains information essential for your Mac programs to compile.

To verify that the PPC compiler is using the proper prefix file, select **Preferences** from the Edit menu. That option is shown in Figure 6.9.

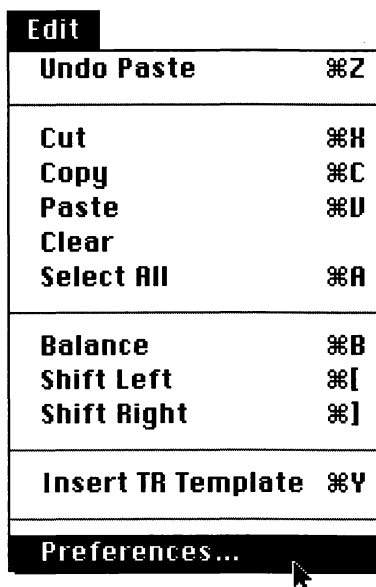
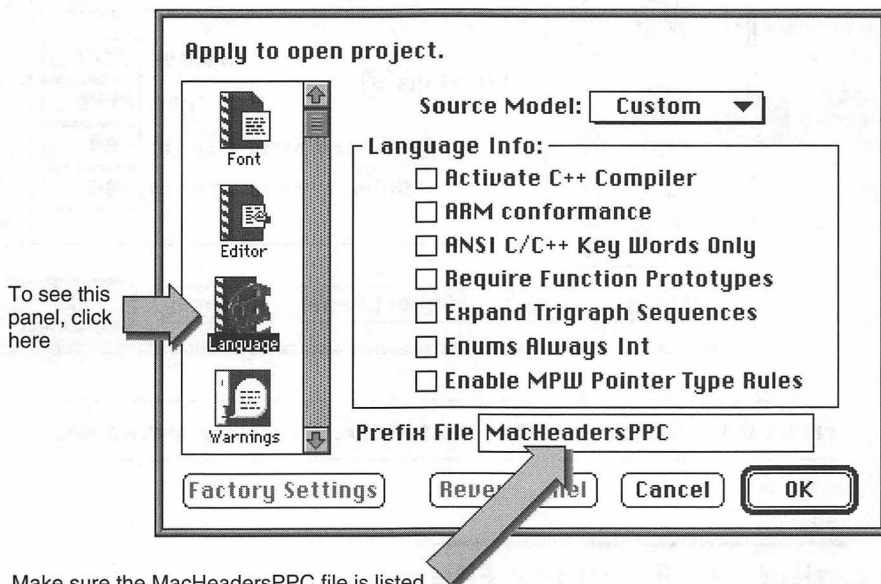


FIGURE 6.9 THE PREFERENCES MENU ITEM IN THE METROWERKS EDIT MENU.

Selecting the **Preferences** menu item will open the dialog box pictured in Figure 6.10. The left side of the dialog box holds a scrollable list of icons. Clicking an icon changes the information displayed to the right of the icon list. Metrowerks calls the different sets of information in this dialog *panels*. To see the panel that allows you to set the project prefix file, click on the **Language** icon in the icon list. The panel that opens will have an edit box that holds the name of the prefix file for the current project. If the MacHeadersPPC file is not named in this edit box, type it in—as shown in Figure 6.10.



Make sure the MacHeadersPPC file is listed here when running the PowerPC version of CodeWarrior

FIGURE 6.10 USING THE PREFERENCES DIALOG BOX TO VERIFY THAT THE PROPER PREFIX FILE IS BEING USED.

Before building a program, you'll want to enter the name the standalone application will be given. If you've dismissed the Preferences dialog box, again select **Preferences** from the Edit menu. Click on the **Project** icon—as shown in Figure 6.11. Verify that the type of the project is set to application. CodeWarrior will give the program the same name as the pro-

Programming the PowerPC

ject—without the “.p” extension. If you want to use a different name, type it in.

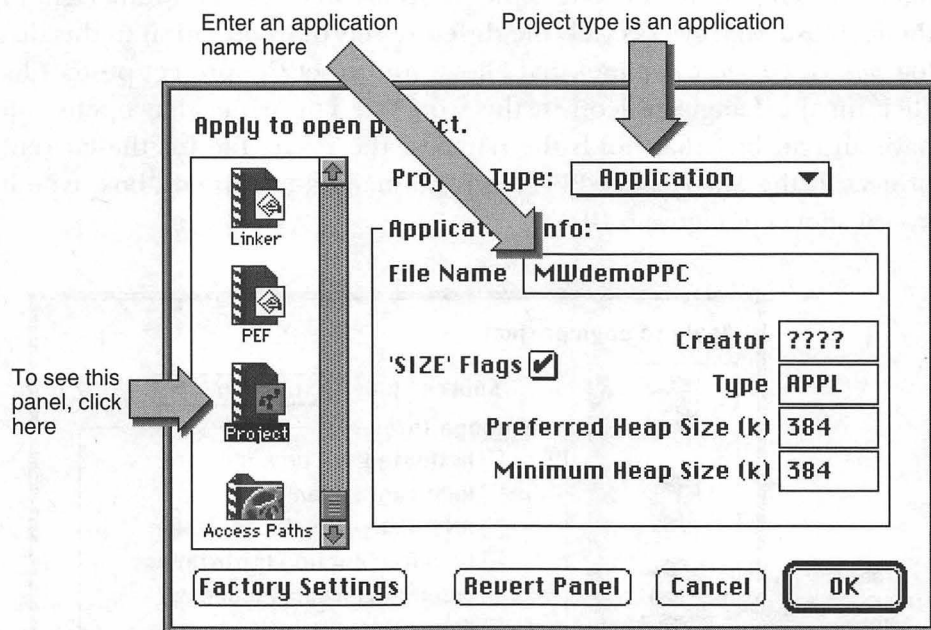


FIGURE 6.11 SUPPLYING A NAME FOR THE FINAL STANDALONE APPLICATION.

Creating the Resource File

The MWdemoPPC program requires a couple of resources. Leave the Metrowerks environment for a moment and launch your resource editor. Create and name a new resource file. Make sure that you're in the folder that holds the Metrowerks project file—the folder I named (6) MW Demo PPC *f*. If you give the resource file the same name as your project, with the “.rsrc” extension appended to the end, CodeWarrior will automatically include the resource file as part of your project. With that in mind I chose the name MWdemoPPC.p.rsrc.



All PowerPC programs also require a ‘SIZE’ resource and a ‘cfrg’ resource. The Metrowerks PowerPC compilers are kind enough to add these resources to your application when you build it. If you’d like more information on these two resource types they’re covered in detail in Chapter 8.

The purpose of this chapter is to cover the basic steps necessary to create a PowerPC application using your compiler—not to create a highly sophisticated program. The MWdemoPPC program fits that description quite nicely. When executed, the MWdemoPPC program does nothing more than display a dialog box with a button in it. Clicking the button closes the dialog box and ends the application. Figure 6.12 shows what the user sees when MWdemoPPC is running.

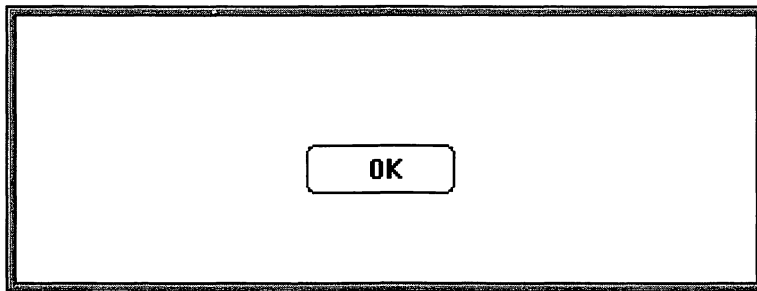


FIGURE 6.12 THE RESULT OF RUNNING THE MWdemoPPC PROGRAM.

A dialog box requires two resources—a ‘DITL’ and a ‘DLOG.’ Figure 6.13 is a view of the ‘DITL’ resource as it looks in the resource editor ResEdit. Figure 6.14 shows the ‘DLOG’ resource. After creating these resources, select **Save** from the File menu, then quit your resource editor.

Programming the PowerPC

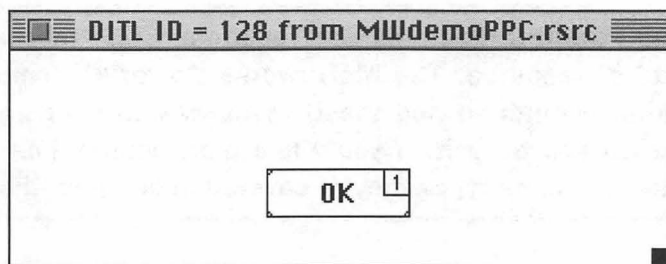


FIGURE 6.13 THE 'DITL' RESOURCE IN THE MWdemoPPC RESOURCE FILE.

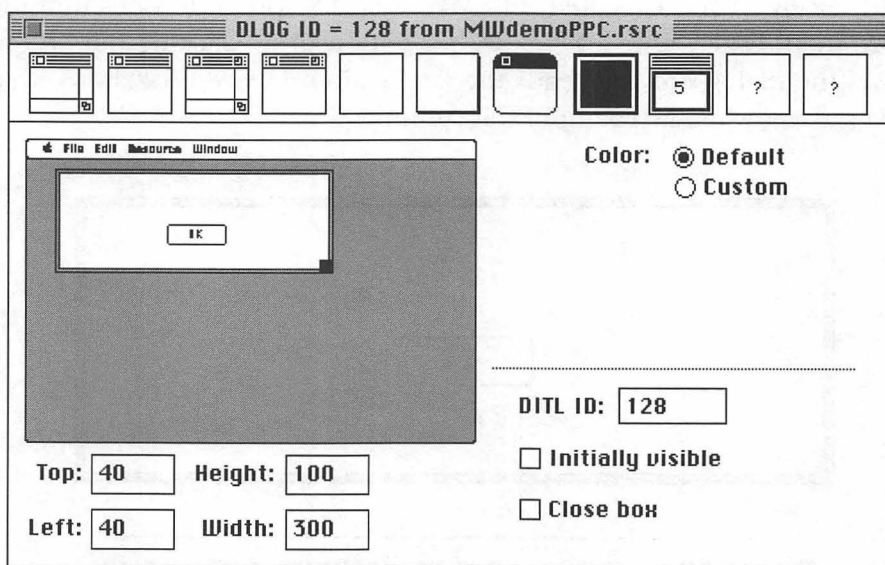


FIGURE 6.14 THE 'DLOG' RESOURCE IN THE MWdemoPPC RESOURCE FILE.

The MWdemoPPC Source Code

The last step before creating the application is to enter the source code. Double-click on the MWdemoPPC.c file to open it. Then type in the following code:

```
//+++++ function prototypes ++++++
void    Initialize_Toolbox( void );
void    Open_Modal_Dialog( void );

//+++++ define directives ++++++
#define    DIALOG_ID        128
#define    OK_BUTTON_ITEM    1

//+++++ main ++++++
void main( void )
{
    Initialize_Toolbox();

    Open_Modal_Dialog();
}

//+++++ initialize the Toolbox ++++++
void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}

//+++++ open a modal dialog ++++++
void Open_Modal_Dialog( void )
{
    DialogPtr    the_dialog;
    short        the_item;
    Boolean      all_done = false;

    the_dialog = GetNewDialog( DIALOG_ID, nil, (WindowPtr)-1L );
    ShowWindow( the_dialog );

    while ( all_done == false )
    {
        ModalDialog( nil, &the_item );

        switch ( the_item )
        {
            case OK_BUTTON_ITEM:
                all_done = true;
        }
    }
}
```

Programming the PowerPC

```
        break;
    }
}
DisposDialog( the_dialog );
}
```

PowerPC compilers require that a function prototype be present for each function you write—the `main()` function being the only exception. After `main()`, the `MWdemoPPC` source code includes two functions. Here are the prototypes for each:

```
void Initialize_Toolbox( void );
void Open_Modal_Dialog( void );
```

After the function prototypes comes the `#define` directives. The two resource IDs are the only `#defines` needed by the program:

```
#define    DIALOG_ID            128
#define    OK_BUTTON_ITEM      1
```

Following the `#define` directives comes `Initialize_Toolbox()`. This function is like any other Toolbox initialization routine you’ve seen in the past:

```
void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}
```

The `Open_Modal_Dialog()` function opens the the program’s dialog box. This same routine—with a few modifications—appears in the following chapter. In the version in Chapter 7 the call to `ModalDialog()` will invoke a filter function.

```
void Open_Modal_Dialog( void )
{
    DialogPtr      the_dialog;
    short          the_item;
    Boolean         all_done = false;

    the_dialog = GetNewDialog( DIALOG_ID, nil, (WindowPtr)-1L );
    ShowWindow( the_dialog );

    while ( all_done == false )
    {
        ModalDialog( nil, &the_item );

        switch ( the_item )
        {
            case OK_BUTTON_ITEM:
                all_done = true;
                break;
        }
    }
    DisposDialog( the_dialog );
}
```

Creating the PowerPC Application

To build an application select **Make** from the Project menu. If you're doing your development on a Power Mac, you can alternately select **Run** from the Project menu. That builds an application and runs it from within the CodeWarrior environment. If you're developing on a 680x0-based Mac, selecting **Run** will cause the compiler to build the application and then attempt to run it—but it won't run it. Instead, you'll see the alert pictured in Figure 6.15. The program won't run because the application will be a PowerPC-only program. While this won't damage the application or the compiler, there's no point in attempting to run the program.

If you're using a Power Mac you can run your new application from the desktop—just double-click on its icon. If you don't own a Power Mac, you won't be able to run the program. If you try to run it from the Finder, you'll see the alert pictured in Figure 6.16.

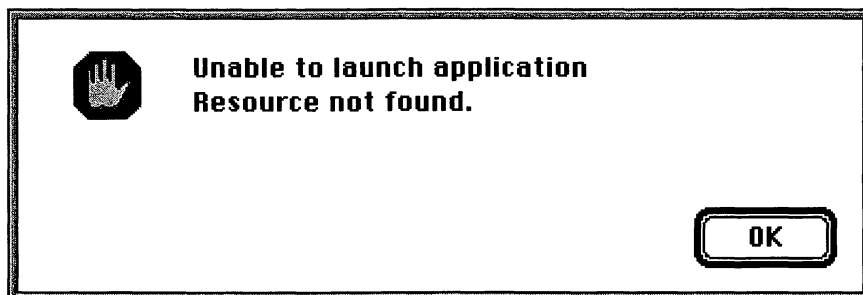


FIGURE 6.15 AN ERROR WILL RESULT WHEN TRYING TO RUN
A POWERPC APPLICATION WHILE DEVELOPING ON A 680x0 MAC.

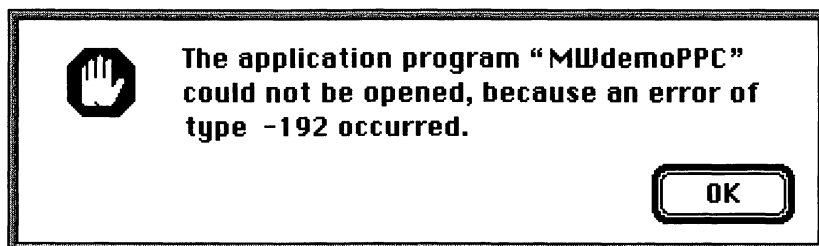


FIGURE 6.16 AN ERROR WILL RESULT WHEN TRYING TO LAUNCH
A POWERPC APPLICATION FROM THE DESKTOP OF A 680x0 MAC.

The Metrowerks MW C/C++ PPC compiler runs on either a 680x0-based Mac or a PowerPC-based Mac. The code it generates, however, can only be run on a Power Mac. Don't blame Metrowerks though—this isn't a flaw in the compiler. A native PowerPC application can only run on a PowerPC-based computer—regardless of the compiler used to generate it. A PowerPC program can be made to run on either platform though. That's done by combining the PowerPC version of the application with a 680x0 version of the same program to form one larger program. This kind of program is called a *fat binary* application and is the topic of Chapter 8.

SYMANTEC'S CROSS-DEVELOPMENT KIT (CDK)

If you own Symantec C++ 7.0 and want to write programs that run native on Power Macs, you'll also need Symantec's Cross-Development Kit, or CDK. The CDK allows you to use either a 680x0-based Mac or a PowerPC-based Mac to develop programs that will run on—and take advantage of—Power Macintoshes.

What the CDK Consists Of

The Symantec CDK uses AppleScripts and ToolServer scripts to automate the process of building a Power Macintosh application. These prewritten scripts allow developers to easily turn their Symantec projects into Power Mac programs.

AppleScript is an integrated scripting system that lets developers write scripts that allow users to perform complex operations with a single click of the mouse button. If you don't have AppleScript, don't worry—the CDK comes with the AppleScript system software extension as well as the Script Editor. And if you don't know how to write scripts, again, don't worry—the CDK comes with prewritten scripts so that you don't have to learn scripting.

ToolServer is a Macintosh Programmer's Workshop (MPW) Shell that allows the use of third-party environments (like the Symantec THINK environment) to execute MPW tools and scripts—even when the MPW compiler isn't present. The MPW tools and scripts necessary to build Power Mac applications are included as a part of the CDK.

When the AppleScript system extension is installed in your System Folder, the menu bar of the THINK Project Manager will have one extra menu appended to it—the AppleScript menu. After creating a project file and writing your source code, you'll make a single menu item selection from the AppleScript menu to build a Power Mac application. This menu item will run the necessary scripts and use the appropriate MPW tools to build the application—without further effort on your part.

Programming the PowerPC

Installing the CDK is simple. First run the **Installer** program that is on Disk #1 of the multiple disk package. In the dialog box that opens, click the **Switch Disk** button to select the hard disk on which you want installation to take place. Then click the **Install** button. The Installer will prompt you to switch disks when necessary.



The CDK discussed here is a developmental release—it's an interim product to allow programmers to create native PowerPC programs while Symantec works on developing a CDK that is more fully integrated into the THINK Project Manager environment. Look for the new version sometime in early 1995.

Installing AppleScript

While AppleScript comes on one of the disks that make up the CDK package, the Installer program doesn't install AppleScript automatically. That's because many people already have AppleScript as part of their installed system software. If you don't already have AppleScript installed on your Mac, you'll want to install it now.

One of the disks that comes with the Symantec CDK is titled AppleScript Setup. Insert this disk in your floppy drive and click on the **Installer** icon. You'll see a dialog box like the one pictured in Figure 6.17. Click the **Install** button. After a few minutes installation will be complete.

The installer adds the AppleScript extension and a few related files to your System Folder. It also creates a folder called AppleScript Utilities and places the Script Editor and Scriptable Text Editor in it. If you aren't familiar with AppleScript, don't be alarmed. As mentioned, the AppleScript-related items will be used when you create a Power Mac application—but not directly by you. Instead, they'll be used by the Symantec environment as you build an application. You won't have to run the AppleScript editor or write any scripts.

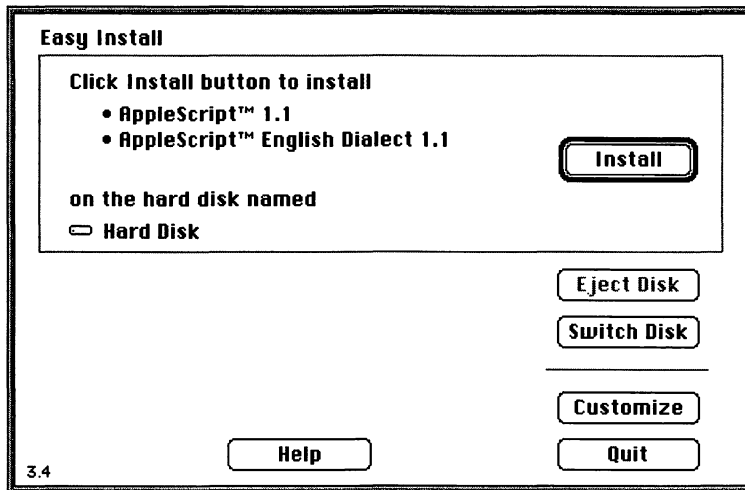


FIGURE 6.17 THE DIALOG BOX FROM SYMANTEC'S APPLESCRIPT INSTALLER.

Using AppleScript to Update ANSI Libraries

When you installed the CDK, the installation made some changes to an ANSI library file. That means the ANSI libraries have to recompile so that they are kept up-to-date. Now that AppleScript is installed, this update is an easy process. Open the **CDK Release Notes** folder that was created during the install of the CDK. It contains a script, shown in Figure 6.18, that will update the ANSI libraries for you. Double-click on the file named **Rebuild ANSI Libraries**.

Double-clicking on the **Rebuild ANSI Libraries** script will launch both the Script Editor and the THINK Project Manager. After a minute or two, you'll see the Script Editor window with a script in it—that's shown in Figure 6.19.

Click on the **Run** button in the Script Editor window to run the script. When you do, a dialog box like the one shown in Figure 6.20 will open. Use the pop-up menu in the dialog to work your way into Standard Libraries folder that's in the Symantec C++ for Macintosh folder. Then click the Choose "**Standard Libraries**" button.

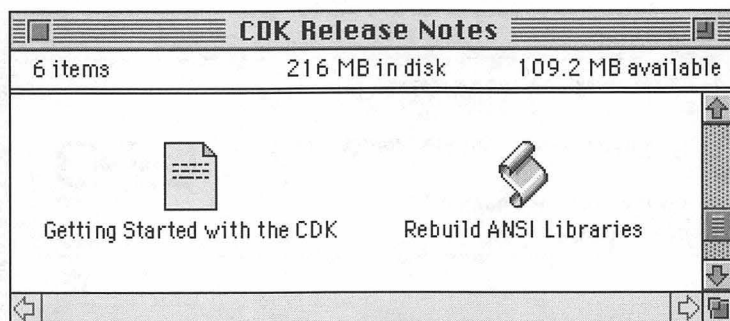


FIGURE 6.18 THE APPLESCRIPT THAT UPDATES THE SYMANTEC ANSI LIBRARIES.

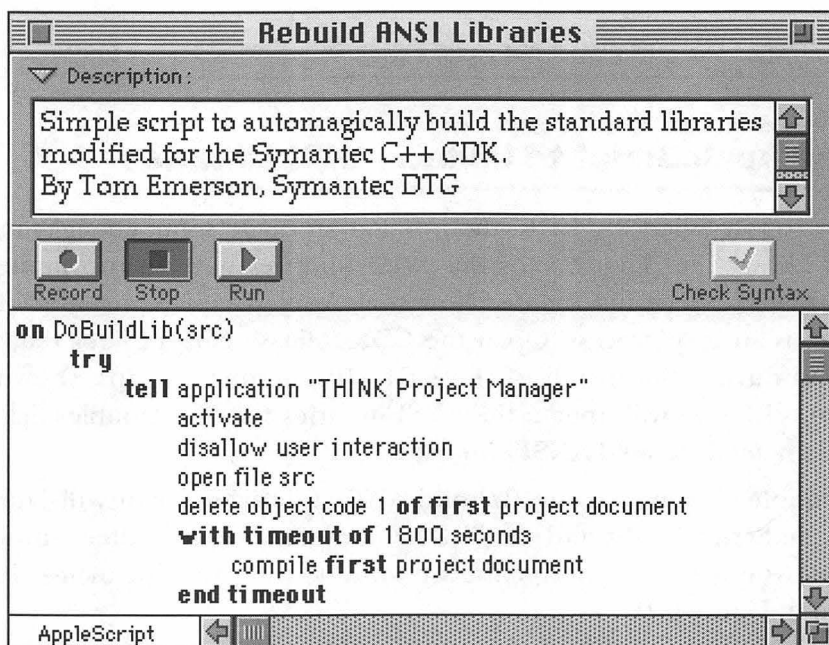


FIGURE 6.19 THE WINDOW THAT RESULTS FROM RUNNING THE REBUILD ANSI LIBRARIES SCRIPT.

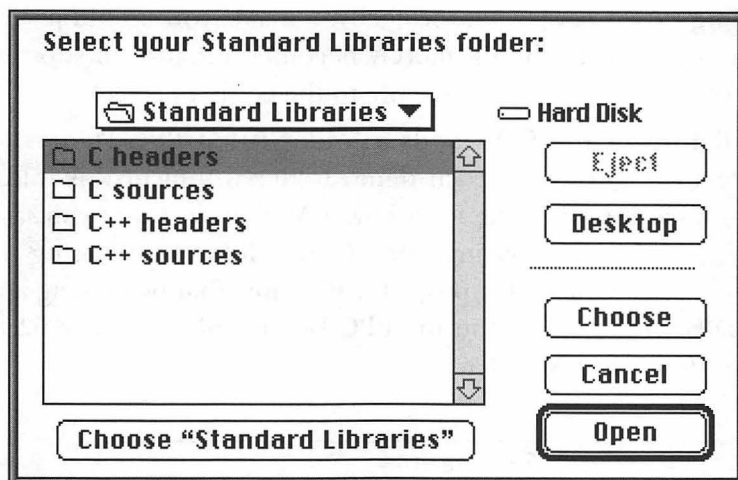


FIGURE 6.20 MOVING TO THE STANDARD LIBRARIES FOLDER WHILE RUNNING THE APPLESCRIPT.

When you click on the **Choose “Standard Libraries”** button, a THINK project will open. The files in the project window will recompile—be prepared for a wait of possibly several minutes. The AppleScript automates this process, so you won’t have to intervene. When the compile is complete, select **Quit** from the File menu. The Script Editor will quit, and you’ll find yourself in the THINK Project Manager. Again select **Quit** from the File menu—this time to exit the Project Manager.

At this point the installation of AppleScript, the installation of the CDK, and the updating of Symantec files is complete. All the steps you’ve taken up to this point were “one-time-only” tasks. Now, it’s time to see how to use the Symantec CDK to build an application that runs native on a Power Mac.

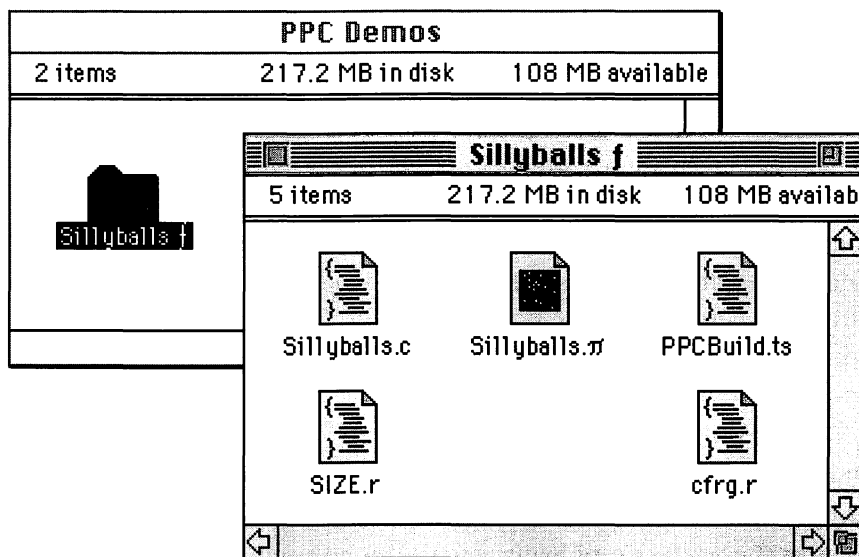
Creating a Folder to Hold Your Power Mac Project

A Symantec CDK project requires several files not found in a “normal” Symantec project. Rather than memorizing the names of these files and

Programming the PowerPC

then adding each to every new project you make, you should just copy an existing CDK project. Then it merely becomes a matter of substituting a new source code file and resource file to the copied project.

Installation of the CDK adds a folder titled PPC Demos to your Development folder. Since each demo folder within in this folder contains the files and project file for a Power Mac project, you can save yourself time and effort by copying one of these folders and using it as the basis of your own Power Mac project. I'll do just that by making a copy of the Sillyballs *f* folder found in the PPC Demos folder. Figure 6.21 shows you the contents of this folder.



**FIGURE 6.21 THE CONTENTS OF THE FOLDER THAT HOUSES
SYMANTEC'S POWERPC DEMO PROGRAM SILLYBALLS.**

To copy the entire Sillyballs *f* folder, click once on it to highlighten it, then choose **Duplicate** from the File menu. The folder will be duplicated, and your PPC Demos window should look like the one shown in Figure 6.22.

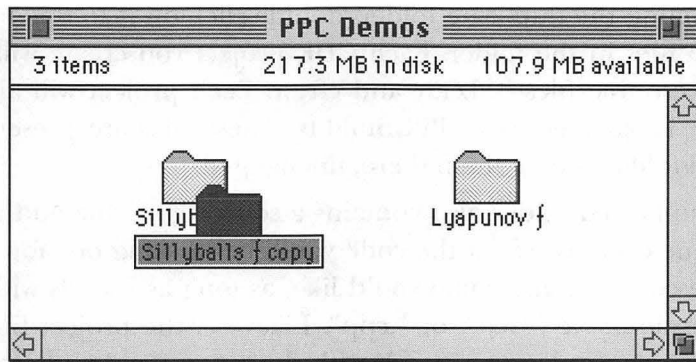


FIGURE 6.22 DUPLICATING THE SILLYBALLS *f* FOLDER.

Rename the duplicate folder. In this chapter I'll walk through the creation of a program I'll call CDKdemoPPC—so I've given the folder a name similar to that. Figure 6.23 shows the new name I've typed in for the folder. The "(6)" refers to the fact that this is a Chapter 6 example. If you want to see what this folder looks like, you'll find it on the disk that was included with the book. If you have the Symantec CDK you can use the project in the (6) CDK Demo PPC *f*—though I'd suggest you try creating your own version so that you become familiar with the process.

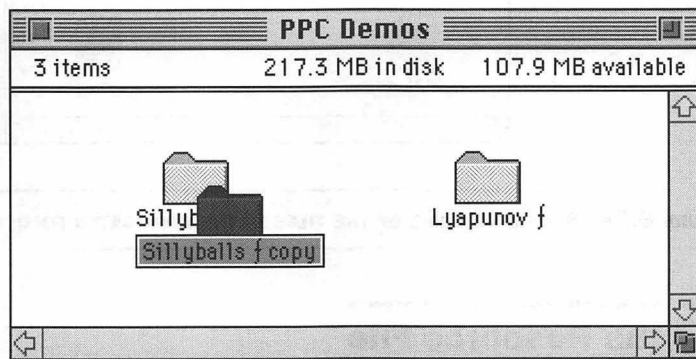


FIGURE 6.23 RENAMING THE DUPLICATED SILLYBALLS *f* FOLDER.

Programming the PowerPC

After renaming the duplicate folder, double-click on it to open it. You'll see several files in the folder. Each CDK project you create will hold at least two resource files—SIZE.r and cfrg.r. Each project will also make use of the ToolServer script PPCBuild.ts. These files are present in the duplicated folder—leave them there, unchanged.

The duplicated folder also contains a source code file and a project file. Rename each to reflect the code you'll be working on. You can give the source code file any name you'd like, as long as it ends with one of three extensions: ".c", ".cp", or ".cpp". Likewise, the project file can be given any name—as long as it ends with the extension ".π". Press the "p" key while holding the Option key down to create the pi symbol. Figure 6.24 shows what the contents of the duplicated folder now look like.

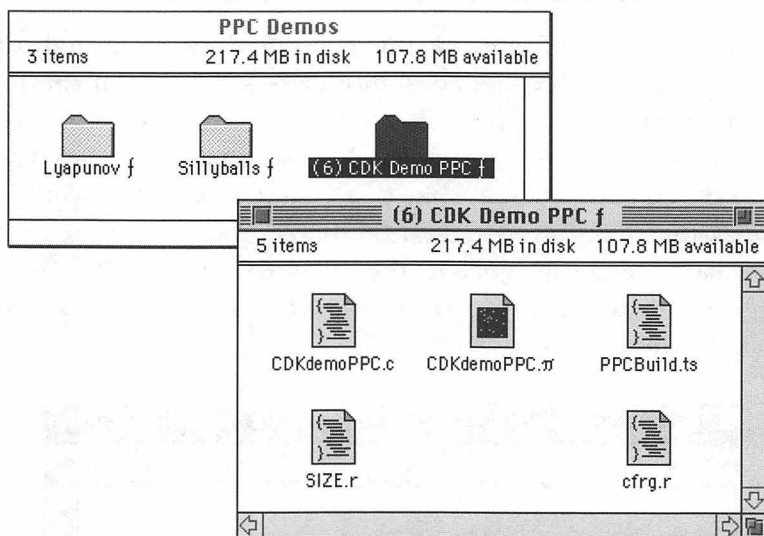


FIGURE 6.24 RENAMING SOME OF THE FILES IN THE DUPLICATED FOLDER.

Creating the Resource File

There's one last step you'll want to take before running the project—you'll want to create a resource file to hold the resources your program

will use. Run your resource editor and create a new resource file. If you give the resource file the same name as your project, with the “.rsrc” extension appended to the end, the THINK Project Manager will automatically include the resource file as part of your project. While you may have been used to doing that in the past, *don’t* do it here—you’ll see why in a moment. Instead, give it any name *but* the project name followed by “.rsrc”. My project is named CDKdemoPPC.π, so I chose a resource name of CDKdemoPPC.rsrc. Note that I omitted the “.π”. If you want your files to match those pictured in the figures of this chapter, give your resource file the same name.

Later, you’ll be explicitly telling the THINK Project Manager which resource files to use with your project. In the past you’ve probably had just a single resource file for one project. For CDK projects, you’ll have more than one. Each project will always include the SIZE.r and cfrg.r resource files that already appear in the file you duplicated. Additionally, your project will include the resource file you make—the one specific to your program. The Project Manager will create a single resource file from these individual files. And the name it gives this final file will be the project name followed by the “.rsrc” extension—that’s why you don’t want to use this name now.

For learning how to use a new environment, simple is better. The CDKdemoPPC program abides by that policy. When executed, the CDKdemoPPC program will display a dialog box with a single item in it—a button. Clicking on the button will close the dialog box and end the running of the program. Figure 6.25 shows what the user will see when he runs CDKdemoPPC.

To support the dialog box that the program displays, the CDKdemoPPC program requires two resources—a ‘DITL’ and a ‘DLOG’. Figure 6.26 shows what the ‘DITL’ resource looks like in ResEdit. Figure 6.27 shows what the ‘DLOG’ looks like. After creating these two resources, select **Save** from the File menu, then quit your resource editor.

Programming the PowerPC

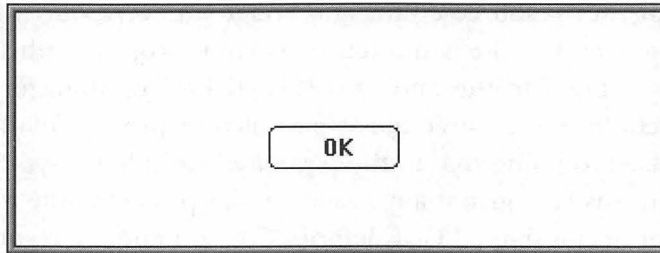


FIGURE 6.25 THE RESULTS OF RUNNING THE CDKdemoPPC PROGRAM.

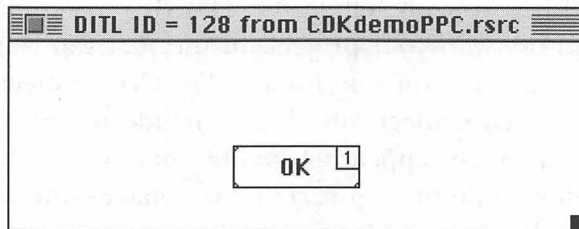


FIGURE 6.26 THE 'DITL' RESOURCE FROM THE CDKdemoPPC RESOURCE FILE.

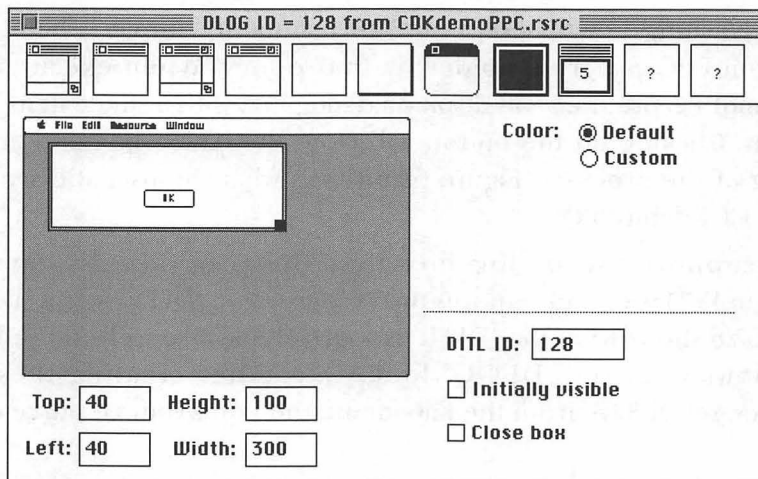


FIGURE 6.27 THE 'DLOG' RESOURCE FROM THE CDKdemoPPC RESOURCE FILE.

After quitting your resource editor, your project folder will have one new file in it. The folder should now look like the one pictured in Figure 6.28.

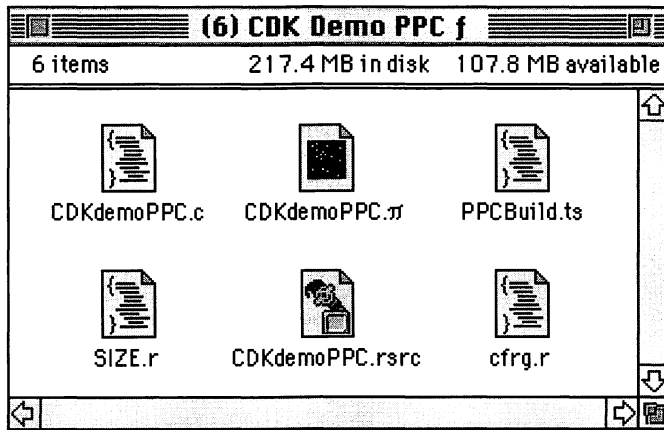


FIGURE 6.28 THE PROJECT FOLDER FOR THE CDKDEMOPPC PROJECT.

Opening the CDK Project

Double-click on the CDKdemoPPC.π project file to start the THINK Project Manager and open the project file. When the project window opens, you'll see ten file names listed in it. The same nine resource, library, and script files that you see should appear in every CDK project you create. Only the one source code file will change. Figure 6.29 emphasizes this.

Figure 6.29 shows the significance of copying an existing CDK project and using it for a new CDK project—you don't have to memorize a lot of file names, or spend time adding files to the project.

The project window doesn't contain the name of the source code file that you'll be using for this project. Select **Add Files** from the Source window to bring up the dialog box that gives you the opportunity to add it. The Source menu is pictured in Figure 6.30.

Programming the PowerPC

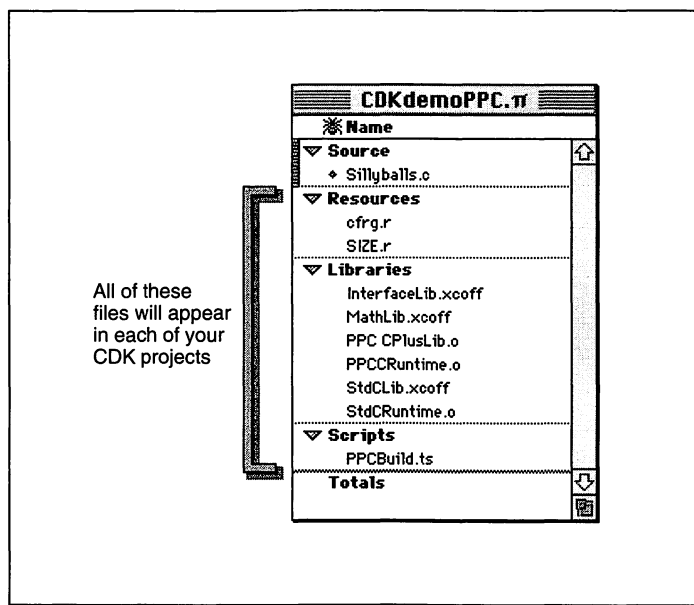


FIGURE 6.29 ALL SYMANTEC CDK PROJECTS CONTAIN THE SAME NINE FILES.

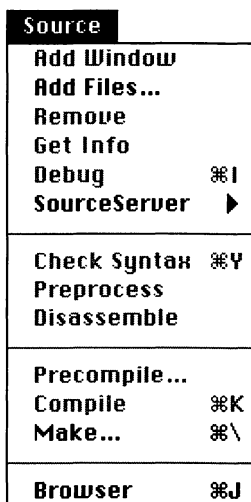


FIGURE 6.30 THE SYMANTEC SOURCE MENU.

Selecting **Add Files** brings up the dialog box pictured in Figure 6.31. If the pop-up menu at the top of the dialog box doesn't show that you're in the (6) CDK Demo PPC *f*, use the menu to move to that folder. Then double-click on the CDKdemoPPC.c file name to add the source code file to the project. Next, double-click on the name of your resource file to add it to the project. Then click the **Done** button.

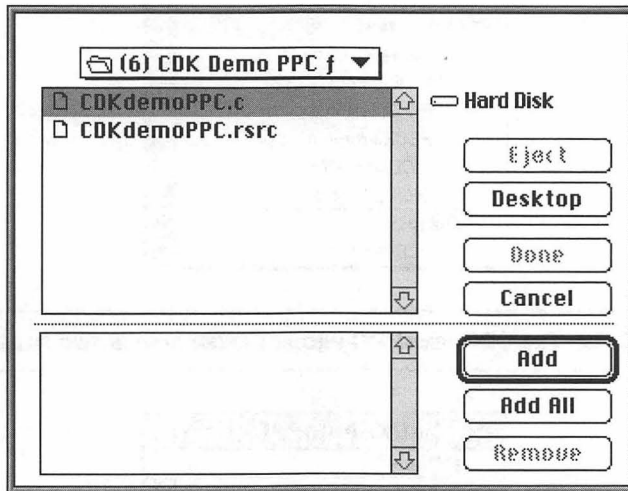


FIGURE 6.31 ADDING FILES TO THE SYMANTEC CDK PROJECT.

After clicking the **Done** button your project window should look like the one pictured in Figure 6.32.

Note that the project file shows the name of the source code file (or files) that appeared in the original project. In this example you'll see that the Sillyballs.c file is in the project. Remove this file from the project by clicking once on the Sillyballs.c file name in the project window, and then selecting **Remove** from the Source menu.

To keep your project window well organized, click once on the CDKdemoPPC.rsrc file name and, with the mouse button still down, drag the file into the segment named Resources. When you've done that your project window will look like the one shown in Figure 6.33.

Programming the PowerPC

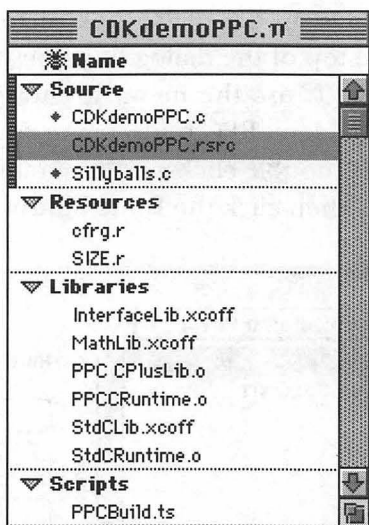


FIGURE 6.32 THE CDKdemoPPC PROJECT AFTER ADDING TWO FILES TO IT.

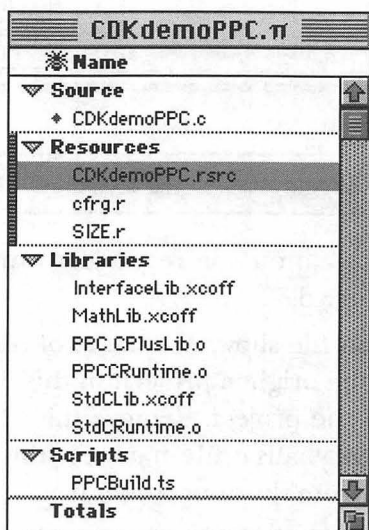


FIGURE 6.33 THE CDKdemoPPC PROJECT AFTER DELETING THE ORIGINAL SOURCE CODE FILE AND MOVING THE RESOURCE FILE.

Required Resources

You'll notice that the project window holds two resource files—other than the one you just added. These two files were duplicated when you copied the Sillyballs folder. Every PowerPC application requires that a 'SIZE' and 'cfrg' be present. Symantec has chosen to create these resources using text descriptions rather than through graphical resource descriptions like those created using ResEdit.

Figure 6.34 shows what the 'SIZE' resource looks like. Double-click on the SIZE.r file in the project window to view its contents. This file can usually be used "as is" from project to project. The only changes you might want to make are to the very last couple of items in the 'SIZE' data structure. These two fields establish the memory partition your application will have—that is, the amount of memory the operating system will allot for your program when the user launches it. There are two numbers related to the partition size. The first is the preferred memory size—the amount of memory the operating system will attempt to secure for the application. The second number is the minimum memory partition. If there is not enough free memory to obtain the preferred size, the operating system will go as low as this second number in its bid to get memory. Figure 6.34 shows that the default value for both of these sizes is 384K bytes. If you feel your program requires more or less memory, change these numbers from "384" to more appropriate values.

If you've used Symantec C++ or THINK C to develop programs for 680x0-based Macs, you're probably familiar with the Set Project Type menu item. It presents you with a dialog box like the one shown in Figure 6.35. For 680x0 applications, this is where you establish the program's partition size. While this dialog box can be used when creating a PowerPC application, the value entered in the Partition (K) edit box will *not* be used. This value will be overwritten by the values used in the 'SIZE' resource. Make sure to make memory partition sizes in the SIZE.r file rather than in this dialog box.

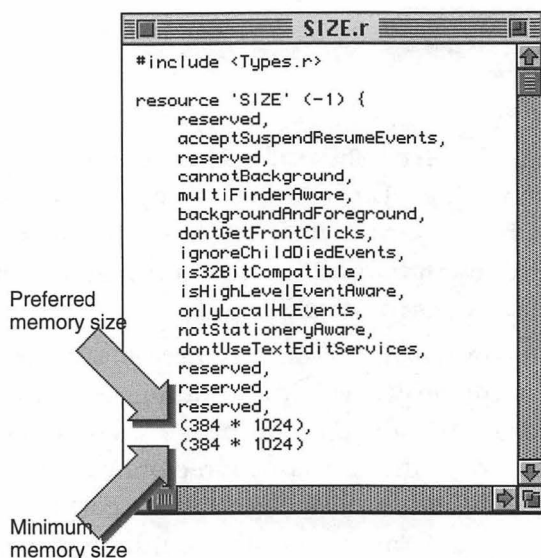


FIGURE 6.34 THE 'SIZE' RESOURCE, VIEWED FROM THE SIZE.R FILE.

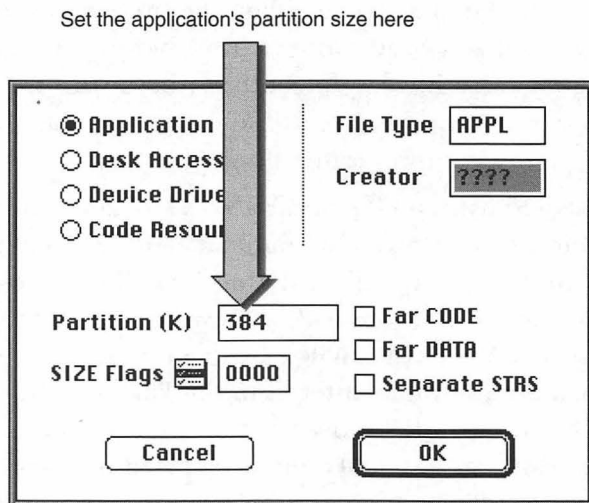


FIGURE 6.35 SETTING AN APPLICATION'S PARTITION SIZE USING SYMANTEC C++ 7.0.

The 'cfrg' resource is unique to PowerPC applications. When a program is launched, the operating system checks the resource fork of the program to see if this resource is present. If it is, the Mac knows it is about to work with a PowerPC program. I'll go into more detail about the 'cfrg' resource in Chapter 8. For now, you'll only have to be aware of one item in the 'cfrg' data structure. The last field of the 'cfrg' holds the name of the application. Double-click on the cfrg.r file and look at this field. Since the file was copied from the Sillyballs folder, it will show the program name as "Sillyballs." Type in the name "CDKdemoPPC"—as I've done in Figure 6.36. For each project you create you'll want to change this one line in the cfrg.r file.

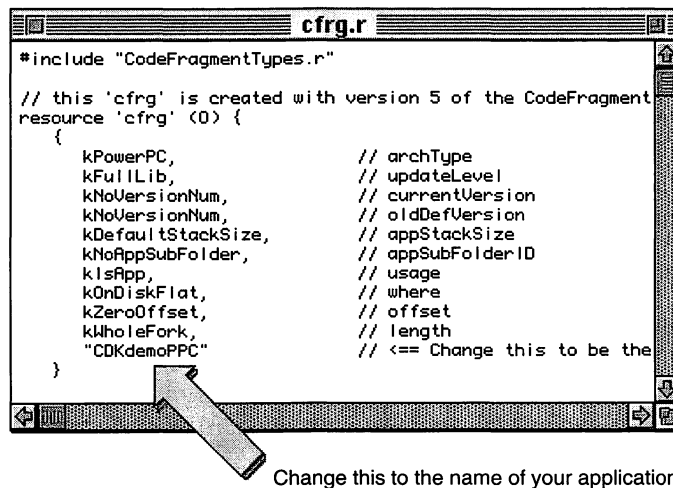


FIGURE 6.36 SETTING THE NAME FOR THE STANDALONE PROGRAM FROM WITHIN THE CFRG.R FILE.

The CDKdemoPPC Source Code

Next, you'll want to edit the source code file. Double-click on the CDKdemoPPC.c file to open it. When you duplicated the entire Sillyballs

Programming the PowerPC

f, a copy of Sillyballs.c was made. At that time you renamed the file CDKdemoPPC.c. Since this file was the Sillyballs.c file, it holds the code for the Sillyballs program. Choose **Select All** from the Edit menu and then press the **Delete** key to delete all the code. Now, type in the following code or, better yet, copy it from the disk that came with this book.

```
//+++++ function prototypes +++++
void    Initialize_Toolbox( void );
void    Open_Modal_Dialog( void );

//+++++ define directives +++++

#define    DIALOG_ID            128
#define    OK_BUTTON_ITEM      1

//+++++ global variables +++++

QDGlobals qd;

//+++++ main +++++

void main( void )
{
    Initialize_Toolbox();

    Open_Modal_Dialog();
}

//+++++ initialize the Toolbox +++++

void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}
```

```
//+++++++ open a modal dialog ++++++

void Open_Modal_Dialog( void )
{
    DialogPtr    the_dialog;
    short        the_item;
    Boolean      all_done = false;

    the_dialog = GetNewDialog( DIALOG_ID, nil,
                              (WindowPtr)-1L );
    ShowWindow( the_dialog );

    while ( all_done == false )
    {
        ModalDialog( nil, &the_item );

        switch ( the_item )
        {
            case OK_BUTTON_ITEM:
                all_done = true;
                break;
        }
    }
    DisposDialog( the_dialog );
}
```

Before creating the PowerPC application, let's take a brief look at the source code for CDKdemoPPC. It's a very simple program, but there are a couple of points worthy of note.

PowerPC compilers are insistent on the inclusion of function prototypes. If you've been lax about their use in the past, now is the time to get consistent. The CDKdemoPPC source code includes two functions (other than `main()`). Here are their prototypes:

```
void Initialize_Toolbox( void );
void Open_Modal_Dialog( void );
```

The only purpose of the program is to display a dialog box. The resource ID of the dialog's 'DLOG' resource is 128. The item number of the only item in the dialog box—the OK button—is 1. These two values appear in the source code as *#define* directives:

Programming the PowerPC

```
#define    DIALOG_ID        128
#define    OK_BUTTON_ITEM    1
```

CDKdemoPPC uses one global variable—the QDGlobals variable `qd`. If you've used Symantec compilers, but haven't compiled for the PowerPC, this declaration may not look familiar to you. Several global variables that most Mac programs make use of are QuickDraw global variables. Two examples are `thePort` and `screenBits`. The five standard patterns, `white`, `ltGray`, `gray`, `dkGray`, and `black`, are also QuickDraw global variables. While in the past it was acceptable to reference these variables directly, you must now preface their names with the name of the data structure which encapsulates them—`qd`. For example, if you used Symantec C++ 6.0 or THINK C 6.0, you may have called `InitGraf()` like this:

```
InitGraf( &thePort );
```

Using Symantec C++ 7.0 and compiling for 680x0-based Macs, you must now call `InitGraf()` like this:

```
InitGraf( &qd.thePort );
```

Note that when compiling with either the Symantec 6.0 or 7.0 compilers, and when compiling for a 680x0 Mac, you need not declare the `qd` data structure. Now, when using Symantec C++ 7.0 and the CDK and compiling for PowerPC-based Macs, you must declare `qd` yourself. So while you'll use QuickDraw globals as you did when compiling 680x0 applications using Symantec C++ 7.0, you'll need to first declare `qd`:

```
QDGlobals qd;
InitGraf( &qd.thePort );
```

The `main()` function's tasks include calling a function to initialize the Toolbox and calling a function to open the modal dialog box. Here's `main()`:

```
void main( void )
{
    Initialize_Toolbox();
```

```
        Open_Modal_Dialog();  
    }
```

The `Initialize_Toolbox()` routine is standard stuff. The only thing worth noting is the call to `InitGraf()`, which is made in the manner just discussed.

```
void Initialize_Toolbox( void )  
{  
    InitGraf( &qd.thePort );  
    InitFonts();  
    InitWindows();  
    InitMenus();  
    TEInit();  
    InitDialogs( 0L );  
    FlushEvents( everyEvent, 0 );  
    InitCursor();  
}
```

The `Open_Modal_Dialog()` function opens the dialog box described by the ‘DLOG’ and ‘DITL’ resources. You’ll see this same routine again, with a couple of modifications, in the next chapter. There the call to `ModalDialog()` will be changed so that it invokes a filter function.

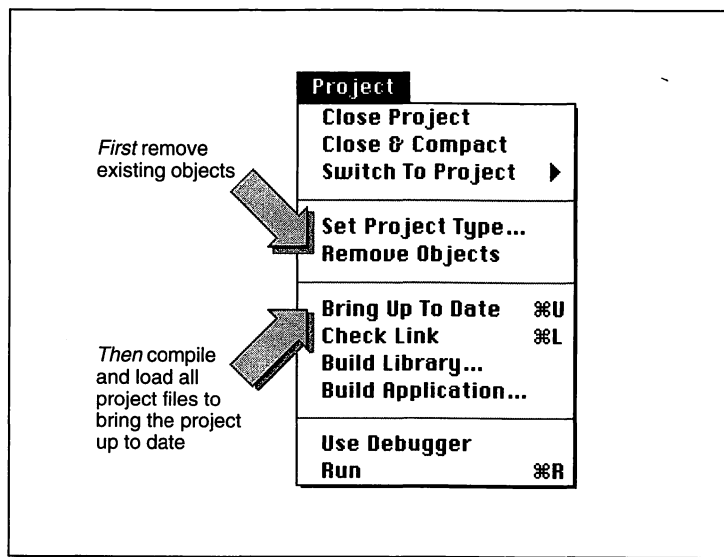
```
void Open_Modal_Dialog( void )  
{  
    DialogPtr      the_dialog;  
    short          the_item;  
    Boolean        all_done = false;  
  
    the_dialog = GetNewDialog( DIALOG_ID, nil, (WindowPtr)-1L );  
    ShowWindow( the_dialog );  
  
    while ( all_done == false )  
    {  
        ModalDialog( nil, &the_item );  
  
        switch ( the_item )  
        {  
            case OK_BUTTON_ITEM:  
                all_done = true;  
                break;  
        }  
    }
```

Programming the PowerPC

```
}  
    DisposDialog( the_dialog );  
}
```

Creating the PowerPC Application

The project contains all of the necessary files, and the source code is written. Normally, the next step would be to bring the project up to date by compiling the new code and loading any libraries that haven't been previously loaded. Since you created this project by copying an existing one, however, there is one step you should take before bringing the project up to date. Select **Remove Objects** from the Project menu. Removing all objects will force the compiler to start from scratch—it will reload all libraries and recompile all of the source code. Figure 6.37 shows the Project menu and the two menu items you should use.



**FIGURE 6.37 REMOVE THE OBJECTS BEFORE BRINGING
A SYMANTEC CDK PROJECT UP TO DATE.**

Selecting **Remove Objects** from the Project menu will result in the display of the alert shown in Figure 6.38. Recompiling and reloading of the files is exactly what you want, so click the **OK** button.

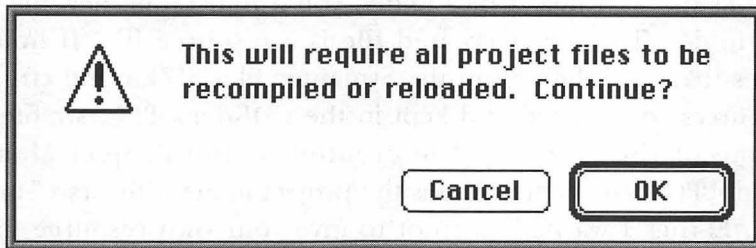


FIGURE 6.38 SYMANTEC ALERTS YOU THAT REMOVING OBJECTS WILL CAUSE FILES TO BE RECOMPILED.

Select **Bring Up To Date** from the Project menu. The THINK Project Manager will load the libraries and compile the source code and resource files. When finished, the folder will contain a new file named `CDKdemoPPC. $\pi$ .rsrc` and a new folder called `(CDKdemoPPC. $\pi$  OBJs)`. Figure 6.39 shows what your project folder should now look like.

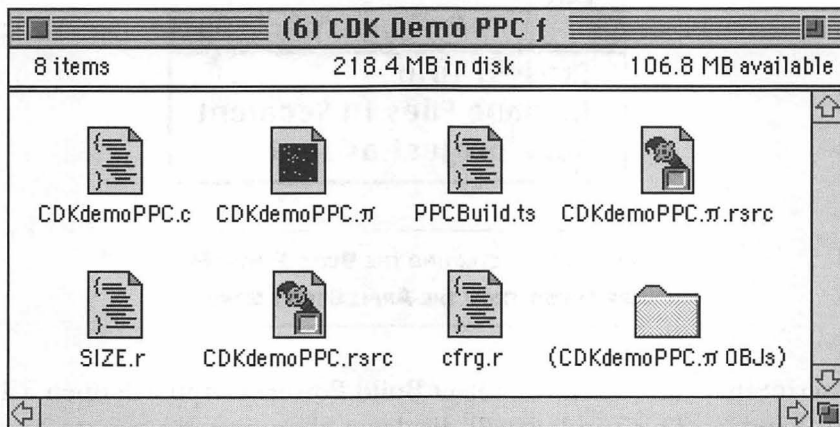


FIGURE 6.39 THE CDK PROJECT FOLDER AFTER THE PROJECT IS BROUGHT UP TO DATE.

Programming the PowerPC

The new folder, named (CDKdemoPPC. π OBJs), contains the object files that result from compiling the files in the project. These files are used by the linker to create the standalone application.

Besides the new object files folder, you'll notice one new file in the project folder. The newly created file is a resource file. It holds the resources that were defined in the Symantec files SIZE.r and cfrg.r, and the resources you created and kept in the CDKdemoPPC.rsrc file. Note the name of the resource file created by the Project Manager: CDKdemoPPC. π .rsrc. The name is the project name with ".rsrc" appended to it. Earlier I warned you not to give your own resource file this name—now you see why. If you had, the THINK Project Manager would have overwritten your resource file with this new one.

Now it's finally time to create the standalone PowerPC application. To do this, you need only run one AppleScript. Fortunately, this script has already been written by Symantec. And, better yet, this one script works for all projects. Click on the **AppleScript** menu that appears to the right of the Windows menu and select **Build PowerPC App**. This menu is shown in Figure 6.40.



FIGURE 6.40 SELECTING THE BUILD POWERPC APP SCRIPT FROM THE APPLESCRIPT MENU.

The script that runs when you select **Build PowerPC App** will open a Tool Status window. This window will display a changing message to let you know at what step in the build the script is at. Figure 6.41 shows the Tool Status window beside the project window.

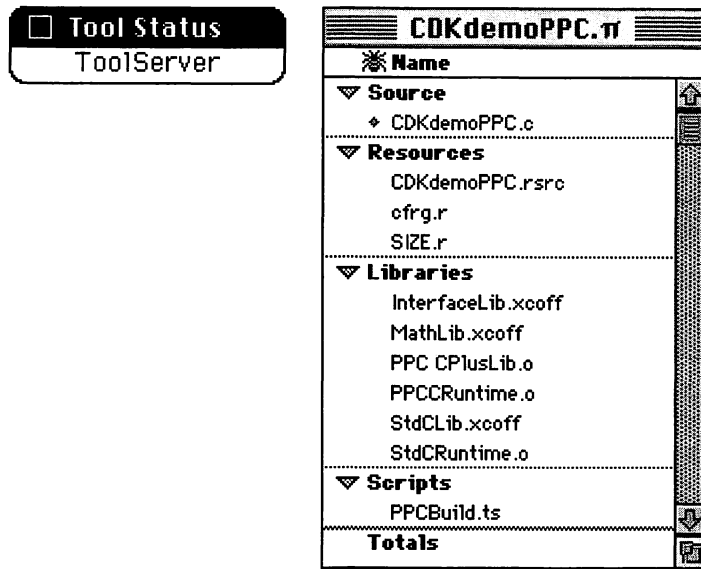


FIGURE 6.41 THE TOOL STATUS WINDOW DISPLAYS EACH STEP THAT TAKES PLACE AS A PROGRAM IS BUILT.

If you forgot to select **Remove Objects** before bringing the project up to date, the AppleScript will fail to complete. You'll see the alert pictured in Figure 6.42.

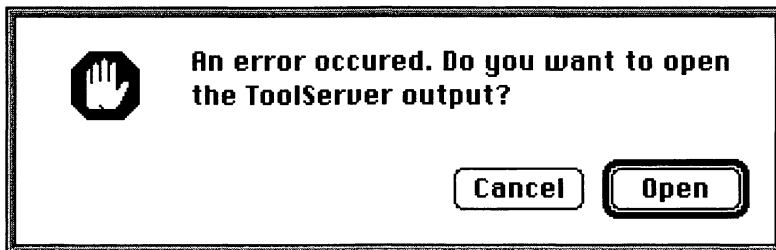


FIGURE 6.42 FORGETTING TO REMOVE THE OBJECTS BEFORE RUNNING THE APPLESCRIPT RESULTS IN AN ERROR.

Programming the PowerPC

If you click on the Open button, you'll see the error message window shown in Figure 6.43. If you're at this point, select **Remove Objects** from the Project menu, then select **Bring Up To Date** from the same menu. Then again select **Build PowerPC App** from the AppleScript menu.

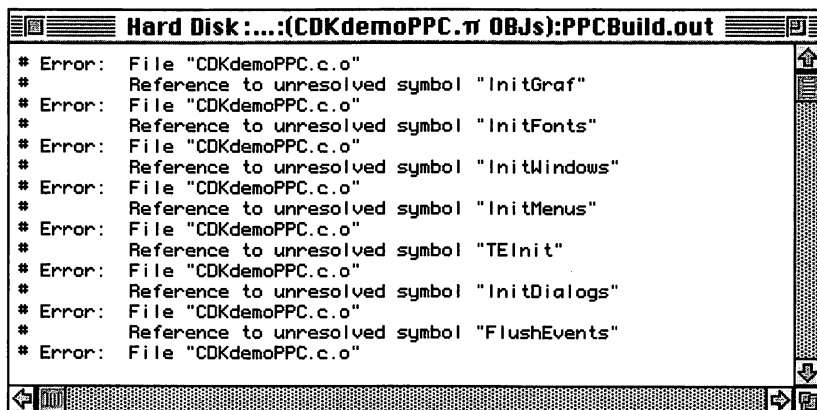


FIGURE 6.43 THE ERROR WINDOW THAT RESULTS FROM THE FAILURE TO REMOVE OBJECTS BEFORE RUNNING THE APPLESCRIPT.

If the build was successful you'll have two new icons in the project folder. One is for the PowerPC application icon. The other is a .SYM files. The .SYM file is used for debugging purposes. Figure 6.44 is a final look at how the project window looks.

If you own a Power Mac you can test the application by double-clicking on its icon to run it. If you don't own a Power Mac, you're out of luck—you'll see the alert pictured in Figure 6.45.

While the Symantec CDK runs on either a 680x0-based Mac or a PowerPC-based Mac, a standalone application that it generates can only be run on a Power Mac. That's not a shortcoming of the Symantec product, however. Any native PowerPC application can only be run on a PowerPC-based computer. It is possible, however, to combine a PowerPC version of an application with a 680x0 version to form one larger program that will run on either platform. This type of program is called a *fat binary*, or *fat application*, and is the topic of Chapter 8.

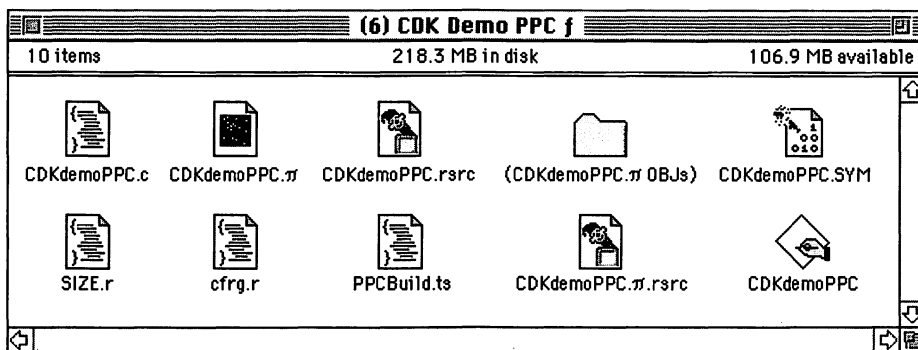


FIGURE 6.44 THE CDK PROJECT FOLDER AFTER THE STANDALONE POWERPC APPLICATION IS BUILT.

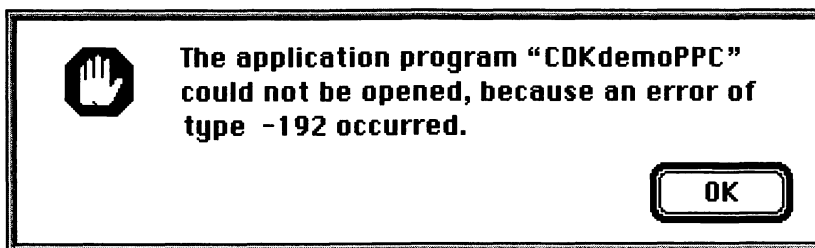


FIGURE 6.45 ATTEMPTING TO LAUNCH A POWERPC APPLICATION FROM THE DESKTOP OF A 680x0-BASED MAC RESULTS IN AN ERROR.

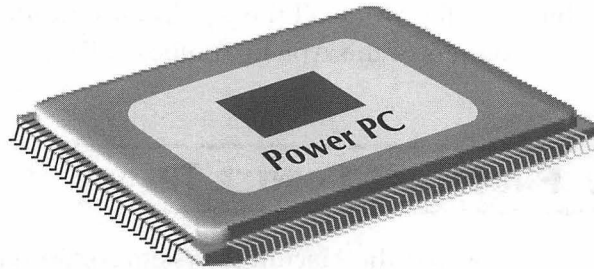
CHAPTER SUMMARY

To create a standalone, native PowerPC application for a Power Mac, you'll need a PowerPC compiler. While other compilers will generate executables that run on both 680x0-based Macs and Power Macs, these programs won't run native on Power Macs. That means that while the program will work on the new family of Macintosh computers, it won't take advantage of the speed of the PowerPC chip.

Programming the PowerPC

Symantec Corporation has long been *the* supplier of Macintosh compilers. Version 7.0 of the Symantec C++ compiler was released close to the time that the Power Mac came to market. While this compiler does create programs that run on the Power Mac, these applications aren't native PowerPC. To create native applications you'll need to get the Symantec Cross Development Toolkit (CDK). This addition to Symantec C++ 7.0 allows you to create native PowerPC applications using either a 680x0-based Mac or a PowerPC-based Mac as your development system.

In 1994 Metrowerks unleashed their entries onto the Macintosh PowerPC compiler market. The Silver edition of their C/C++ compiler generates native PowerPC applications using either a 680x0-based Mac or a Power Mac as the development system. The Gold edition includes two separate compilers. Both use either a 680x0-based Mac or a Power Mac as the development system. The first of the two compilers generates 680x0-only applications, while the second generates PowerPC-only programs.



CHAPTER 7

UNIVERSAL PROCEDURE POINTERS

The Mixed Mode Manager makes it possible for an application to contain a mix of 680x0 code and native PowerPC code. Functions that you write may make calls to Toolbox routines that have or haven't been ported—with no extra work on your part. The Mixed Mode Manager does an incredible job of keeping track of what mode the Mac should be in at all times—PowerPC or 68LC040 emulator. Still, there will be a few occasions when you will have to give this manager an assist. You'll do that by including a special pointer type in your source code—the universal procedure pointer, or `UniversalProcPtr`. This pointer type, which is new to Mac programming, enables you to pass the address of one of your own functions to a Toolbox routine. In the past, this was done through the use of a procedure pointer, or `ProcPtr`.

Several Toolbox routines—`ModalDialog()` and `SetDItem()` being the most notable—make use of `ProcPtr`s on the 680x0-based Macs. On the PowerPC-based Macintosh these routines use `UniversalProcPtr`s

Programming the PowerPC

instead. This chapter will provide all the details on what this pointer type is, and how to use pointers of this type in Toolbox calls.

UNIVERSAL PROCEDURE POINTER THEORY

In Chapter 4 you saw that the Macintosh system software is a mixture of old 680x0 functions and new, native PowerPC functions. While Apple has ported the most frequently called Toolbox routines to native code, the majority of the Toolbox routines have not been ported. The Mixed Mode Manager is aware of which routines are 680x0 code and which are PowerPC code. That frees you, the programmer, from worrying about any mixed mode details when you add a call to a Toolbox routine to your code.

The Mixed Mode Manager is fully capable of properly routing calls to both ported and nonported Toolbox routines. It cannot, however, handle calls to functions which it knows nothing about. Not, anyway, without a little help from the programmer.

Procedure Pointers and the 680x0 Processor

Programmers of all pre-PowerPC Macintosh computers who use modal (nonmovable) dialog boxes will be familiar with a line of code such as the following:

```
ModalDialog( nil, &the_item );
```

If the user of a program clicks the mouse button while over an enabled item in a modal, or nonmovable, dialog box, the Toolbox function `ModalDialog()` will report this fact to the program. `ModalDialog()` will then take control. For example, if the user clicked on a dialog button, `ModalDialog()` is responsible for changing the highlighting of the button.

While the first parameter to `ModalDialog()` is most often `nil`, it doesn't have to be. If the programmer wants to perform some special

Chapter 7 Universal Procedure Pointers

action when a dialog box item is clicked-on, this first parameter should be the name of a special filter function. If a function name does appear as the first parameter, that function will be invoked before `ModalDialog()` has the opportunity to take action. If I wrote a filter function named `My_Filter()`, then a call to `ModalDialog()` would look like the one shown here:

```
ModalDialog( My_Filter, &the_item );
```

Each time `ModalDialog()` was called, the filter function would first execute—as shown in Figure 7.1.

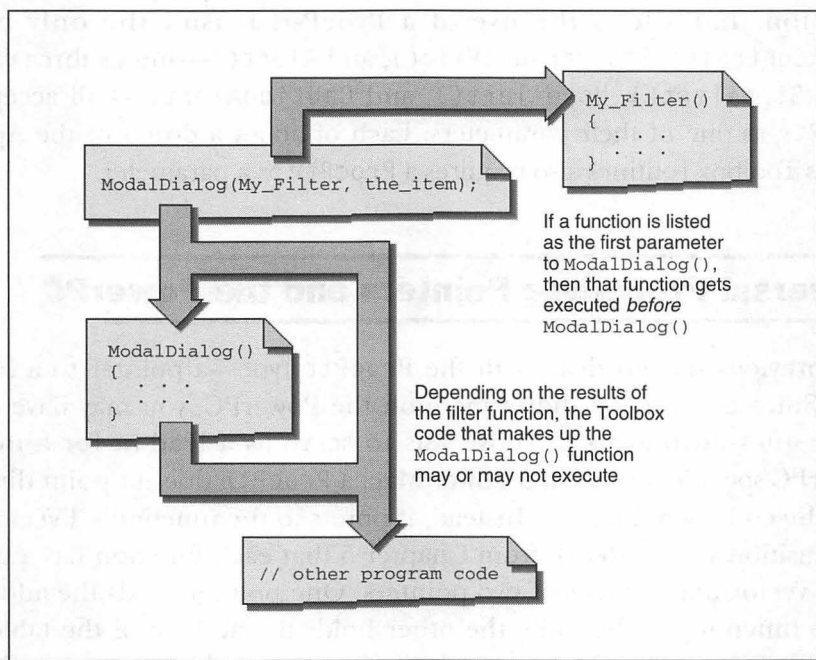


FIGURE 7.1 A FILTER FUNCTION ALLOWS THE PROGRAMMER TO ADD TO THE FUNCTIONALITY OF `ModalDialog()`.

Filter functions are handy for—naturally enough—filtering. In the case of `ModalDialog()`, a filter function is often used to determine if the

Programming the PowerPC

user pressed a Command key combination. If the user did, the filter function handles it in its own way—and the Toolbox code that makes up `ModalDialog()` is skipped. If the user wasn't pressing the Command key when an item was clicked on in the dialog box, the filter function doesn't handle the action, and the `ModalDialog()` code does.

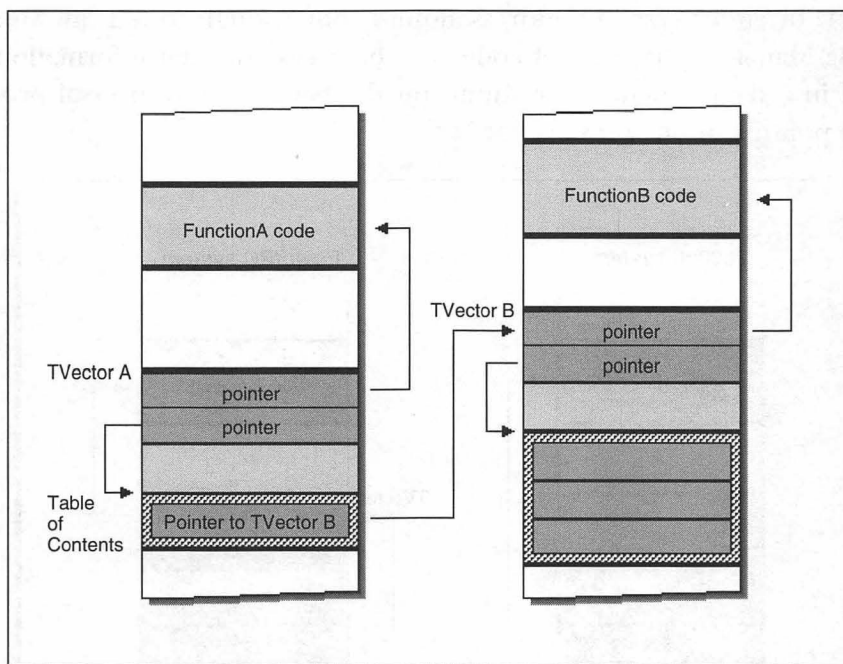
The first parameter to `ModalDialog()`—whether `nil` or a function name—is actually a pointer. In the case of `nil`, it is of course a `nil` pointer. In the case of a function name, it is a pointer to the code that makes up that function. On the Macintosh, this type of pointer is called a `ProcPtr`—a pointer to a function.

While `ModalDialog()` may be the most notable example of a Toolbox function that allows the use of a `ProcPtr`, it isn't the only one. `TrackControl()`, `StandardGetFile()`, and `Alert()`—and its three variations, `StopAlert()`, `NoteAlert()`, and `CautionAlert()`—all accept a `ProcPtr` as one of their parameters. Each of about a dozen of the Apple Events Toolbox routines also requires a `ProcPtr` as a parameter.

Universal Procedure Pointers and the PowerPC

The previous section dealt with the `ProcPtr` type—a pointer to a function. Since that section didn't mention the PowerPC, you may have correctly guessed that its purpose was to serve as a lead-in for a more PowerPC-specific topic. On a Power Mac, a `ProcPtr` doesn't point directly to the code of a function. Instead, it points to the function's `TVector`—its transition vector. Recall from Chapter 5 that each function has a transition vector that consists of two pointers. One pointer holds the address of the function's code, while the other holds the address of the table of contents for the fragment in which the function resides. Together these two pointers keep track of the function and any other routines that the function may call. Chapter 5 closed with a figure that summarized `TVectors`—it's reprinted here as Figure 7.2.

Figure 7.3 compares a `ProcPtr` used on a 680x0-based system with a `ProcPtr` used on a PowerPC-based system.



**FIGURE 7.2 A TVECTOR CONTAINS TWO POINTERS—
ONE TO A FUNCTION'S CODE, THE OTHER TO THE FRAGMENT'S TOC.**

There is still another difference between ProcPtrs used on different processor-based systems—how they are used by the programmer. You've seen that on a 680x0-based Mac, a ProcPtr can be used directly. ModalDialog(), for instance, allows a function's name to be used as the first parameter and to serve as a ProcPtr:

```
ModalDialog( My_Filter, &the_item );    // My_Filter is a  
                                         ProcPtr
```

On a PowerPC-based Mac, things are more complicated. Because instructions from two different instruction sets (680x0 and PowerPC instructions) are used within a single program, the Mixed Mode Manager needs help in handling calls that involve ProcPtrs. A ProcPtr is an address (680x0-based systems) or a pointer to an address (PowerPC-based sys-

Programming the PowerPC

tems). In either case, an address alone is not enough to tell the Mixed Mode Manager what type of code is to be accessed. That information is held in a data structure not found on the 680x0—the universal procedure pointer, or `UniversalProcPtr`.

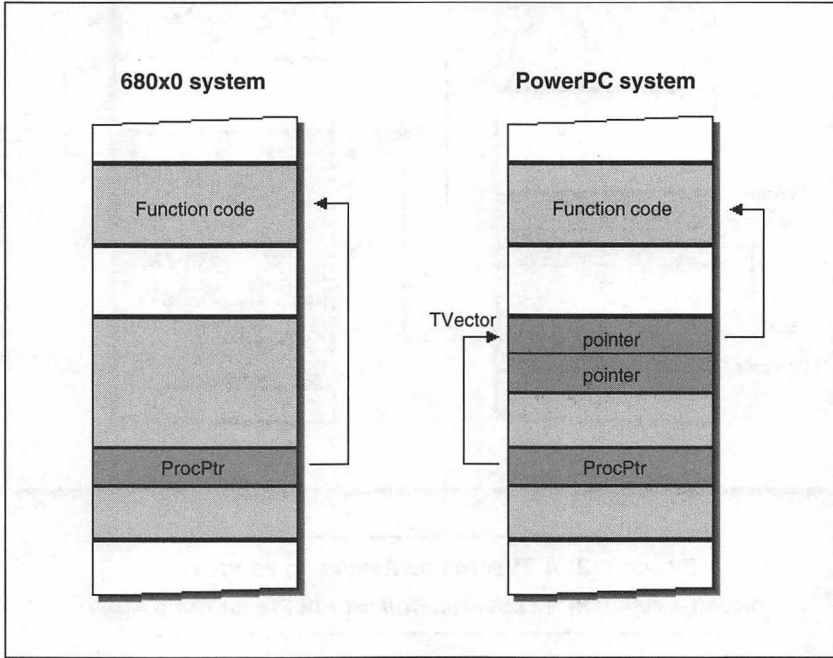


FIGURE 7.3 COMPARISON OF `ProcPtr`s ON A 680x0-BASED SYSTEM AND A POWERPC-BASED SYSTEM.



`UniversalProcPtr`s play a big part in porting 680x0 code to the Power Mac. In fact, the conversion of function pointers of type `ProcPtr` to type `UniversalProcPtr` will probably represent your largest porting task—that's why I've dedicated an entire chapter to the subject. This chapter—as well as code in subsequent chapters—will provide plenty of examples.

Chapter 7 Universal Procedure Pointers

On a PowerPC-based Mac, a UniversalProcPtr (or UPP) is used much as a 680x0-based Mac uses a ProcPtr. The UniversalProcPtr just yields more information than a ProcPtr. The UniversalProcPtr is a pointer to still another data structure—a routine descriptor, or RoutineDescriptor. The RoutineDescriptor holds information about the function, or routine, that is to be invoked. One of the pieces of information in the RoutineDescriptor is a flag that specifies the instruction set architecture of the routine. That is, whether the routine can be handled directly by the PowerPC chip or whether the Mixed Mode Manager must pass instructions to the 68LC040 emulator for preliminary processing. Another field in the RoutineDescriptor is a ProcPtr. This ProcPtr leads, via the TVector, to the functions code. Figure 7.4 illustrates this.

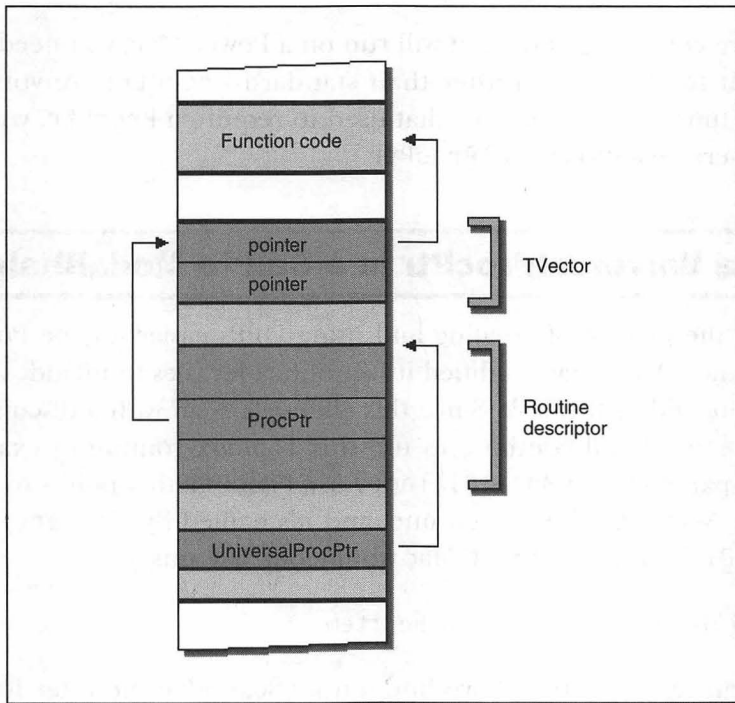


FIGURE 7.4 A UniversalProcPtr INDIRECTLY LEADS TO THE CODE THAT MAKES UP A FUNCTION.

Programming the PowerPC

Don't despair if Figure 7.4 gives you the feeling that things appear to be getting very complicated—I won't be adding to the number of data structures and pointers in that figure.



And, of more significance yet, you won't be responsible for organizing TVectors, RoutineDescriptors, or any other pointer or data structure regarding function calls via ProcPtrs. Rather, you'll simply create a UniversalProcPtr and let the compiler organize things.

USING UNIVERSALPROCPTRS

If you're compiling code that will run on a Power Mac, you need to use UniversalProcPtrs rather than standard ProcPtrs. Anytime you invoke a function in a manner that used to require a ProcPtr, you must now first create a UniversalProcPtr.

Using a UniversalProcPtr in a Call to ModalDialog()

To make the process of creating and using UPPs easier for the PowerPC programmer, Apple has modified its set of header files to include a set of macros that aid in this task. Since this chapter began with a discussion of ModalDialog(), I'll continue to use this Toolbox routine in examples. The first parameter to ModalDialog() is a ProcPtr that points to a filter function. Assuming I've written one, and it's named My_Filter(), a call to ModalDialog() on a 680x0 Mac would look like this:

```
ModalDialog( My_Filter, &the_item );
```

As you can see from the above line, on a 680x0 Mac the filter function name serves as the ProcPtr. On a Power Mac, things are quite different.

To work with a filter function on a Power Mac I need to first declare a UniversalProcPtr variable and then create a routine descriptor for the filter function. Here's how that's done:

Chapter 7 Universal Procedure Pointers

```
ModalFilterUPP my_filter_UPP;  
my_filter_UPP = NewModalFilterProc( My_Filter );
```

Then, rather than use the name of the filter function—as done on a 680x0-based Mac, I pass the UniversalProcPtr to ModalDialog():

```
ModalDialog( my_filter_UPP, &the_item );
```

The above example is worthy of closer examination. First, we'll look at the declaration of the variable `my_filter_UPP`. With the advent of the universal headers, Apple has defined a whole new set of data types that are to specifically be used when working with universal procedure pointers. `ModalFilterUPP` is one such data type. In the universal header file `Dialogs.h`, you'll find this definition:

```
typedef UniversalProcPtr ModalFilterUPP;
```

So it turns out the `ModalFilterUPP` is nothing more than a `UniversalProcPtr`.

Declaring a `UniversalProcPtr` variable doesn't create a routine descriptor for a function—as shown in Figure 7.5. For that, you need to call one of the many macros that creates a routine descriptor, examines a function, and fills in the fields of the routine descriptor accordingly. These macros are defined in the universal header files. To set up the `UniversalProcPtr` for my filter function that is to be used by `ModalDialog()`, I'd call the `NewModalFilterProc()` macro:

```
my_filter_UPP = NewModalFilterProc( My_Filter );
```

The call to `NewModalFilterProc()` creates a routine descriptor with information specific to the function whose name appears as the single function parameter. When the call to `NewModalFilterProc()` is complete, a `UniversalProcPtr` will be returned. Variable `my_filter_UPP` is then associated with the `My_Filter()` function. That's shown in Figure 7.6. Now, in place of the function name (as is done in 680x0 development), use the `UniversalProcPtr` variable:

```
ModalDialog( my_filter_UPP, &the_item );
```

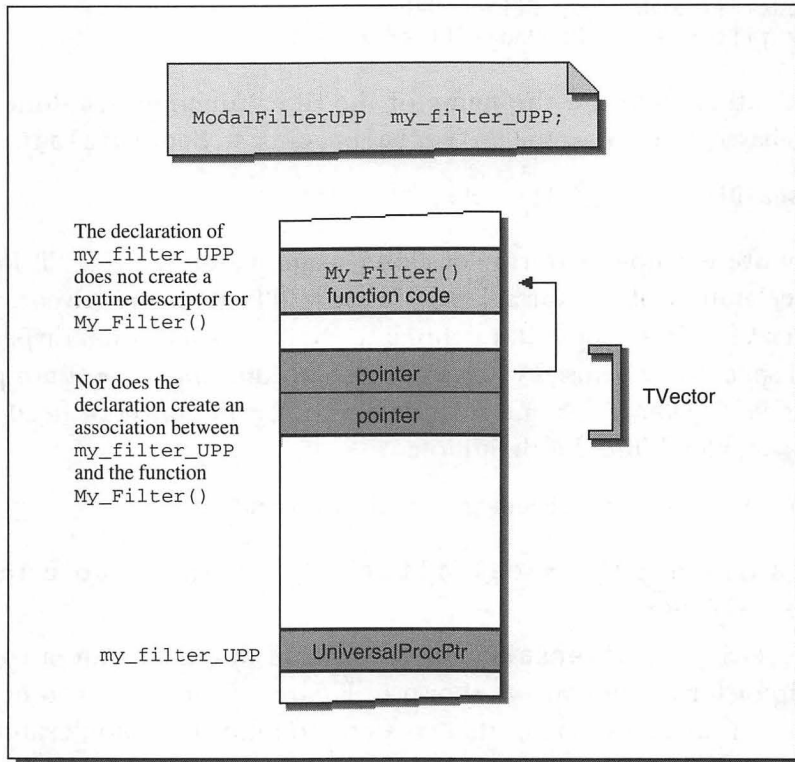


FIGURE 7.5 THE DECLARATION OF A UniversalProcPtr DOES NOT ASSOCIATE THAT POINTER WITH A FUNCTION.

To summarize, the 680x0 method of passing a pointer to a function is to simply use the function's name in place of the parameter that is of type `ProcPtr`:

```
ModalDialog( My_Filter, &the_item );
```

The PowerPC method of passing a pointer to a function is to first declare a `UniversalProcPtr` (of the type specific to the Toolbox routine in which it will be used). Next, create a routine descriptor and set up the UPP via one of the Apple-implemented macros designed to do just that. Finally, use the UPP as the parameter that is to serve as the pointer to the function:

Chapter 7 Universal Procedure Pointers

```
ModalFilterUPP my_filter_UPP;  
my_filter_UPP = NewModalFilterProc( My_Filter );  
  
// open dialog here  
  
ModalDialog( my_filter_UPP, &the_item );
```

When you're through with the UPP, dispose of it just before you leave the function in which it was allocated:

```
DisposeRoutineDescriptor( my_filter_UPP);
```

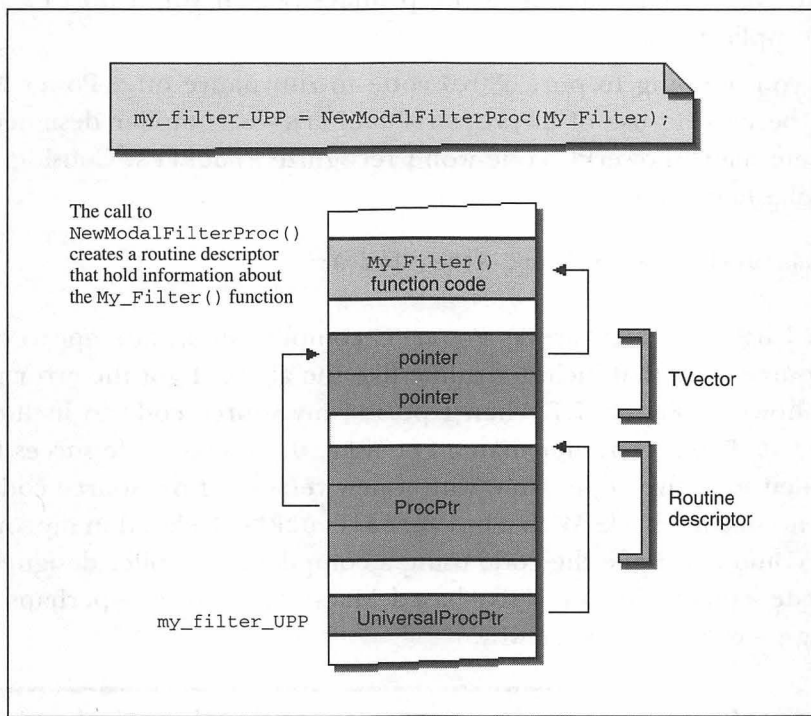


FIGURE 7.6 AN APPLE-DEFINED MACRO SUCH AS `NewModalFilterProc()` CREATES A ROUTINE DESCRIPTOR AND RETURNS A `UniversalProcPtr`.

Later in this chapter you'll see some of the other Apple macros that create routine descriptors.

How the Compiler Chooses Between ProcPtr and UniversalProcPtr

Almost all applications that were written before the arrival of the Power Macs—those programs designed for 680x0-based Macs—will run without modification on a Power Mac. This type of program won't, however, take advantage of the speed of the PowerPC chip. That's because the application will spend most of its time in emulation mode, with instructions being handled by the 68LC040 Emulator. Modifying programs of this type to run in native mode is the primary reason you'll port existing 680x0 applications.

If you're going to port 680x0 code to run native on a Power Mac, you'll be making use of `UniversalProcPtr`s—a compiler designed to generate native PowerPC code won't recognize `ProcPtr`s. Consider the following line of code:

```
ModalDialog( My_Filter, &the_item );
```

When I used the Metrowerks PowerPC compiler in an attempt to compile source code that included a line like the above, I got the error message shown in Figure 7.7. When I ported my source code to include a `UniversalProcPtr` rather than a `ProcPtr`, the source code successfully compiled to native code. Now, with a new version of my source code, is there no turning back? With a `UniversalProcPtr` declared in my source code, could I compile the code using a compiler a compiler designed to generate executables for 680x0-based Macs? The answer—perhaps surprisingly—is “yes.” Let's see why.



Source code for a small application that uses a `UniversalProcPtr` appears later in this chapter.

If you program in C or C++, you may be familiar with the CodeWarrior compilers by Metrowerks—they were covered in detail in the previous chapter. There are two versions of the CodeWarrior C/C++ compiler.

Chapter 7 Universal Procedure Pointers

One is named MW C/C++ 68K, and the other is named MW C/C++ PPC. The first generates executables that run on 680x0-based Macs, the second creates executables for PowerPC-based Macintoshes.

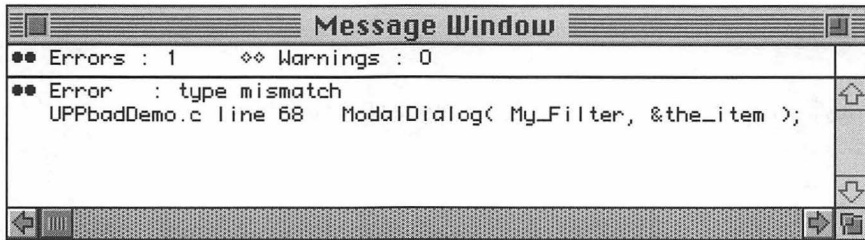


FIGURE 7.7 POWERPC COMPILERS ALERT YOU WHEN YOU ATTEMPT TO USE A ProcPtr IN PLACE OF A UPP.



NOTE

When used in conjunction, the two separate compilers generate code that can be combined to create a *fat binary*—a program that can be run on either a 680x0 or Power Mac. Chapter 8 provides the details on that.

If I write C or C++ code that makes use of a UniversalProcPtr, it will of course compile on the MW C/C++ PPC compiler. But it will also compile on the MW C/C++ 68K compiler. That's because the Metrowerks compilers, like version 7.0 of the Symantec compiler, makes use of the universal header files. By using *conditional preprocessor directives*, the new header files can be used by 680x0 compilers or PowerPC compilers.

You're certainly familiar with preprocessor directives like `#include` and `#define`. Conditional compilation directives are another type of preprocessor directive. They allow a file to be compiled in more than one way. By using preprocessor directives like `#if` and `#else`, the compiler can be forced to compile or ignore blocks of code according to the conditions at the time the compile takes place. In the universal version of the `Dialogs.h` header file you'll find conditional compilation directives that cause `ModalFilterUPP` to be defined in two different ways. Here's an edited version of a part of `Dialogs.h`:

Programming the PowerPC

```
#if USESROUTINEDESCRIPTORS

typedef UniversalProcPtr ModalFilterUPP;
. . .
. . .
#else

typedef ModalFilterProcPtr ModalFilterUPP;
. . .
. . .
#endif
```

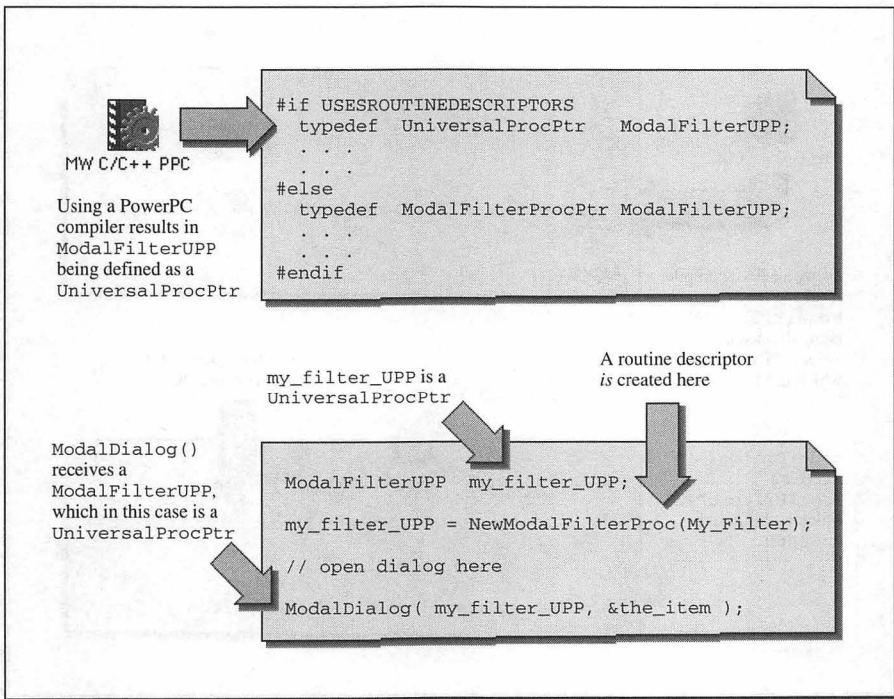
You can see from the above that if the flag `USESROUTINEDESCRIPTORS` is present, `ModalFilterUPP` will be defined as a `UniversalProcPtr`. If it isn't, then `ModalFilterUPP` will be defined as a `ModalFilterProcPtr`. A `ModalFilterProcPtr` is a type of `ProcPtr`, and can be used by 680x0-based compilers. `Dialogs.h` also uses conditional compilation directives to define what `NewModalFilterProc()` looks like. If compiling takes place using a compiler that is to generate PowerPC code, `NewModalFilterProc()` is defined such that it will create the necessary routine descriptor. If the compiling takes place using a compiler that generated 680x0 code, then `NewModalFilterProc()` won't create a routine descriptor. How the two different Metrowerks compilers handle procedure pointers is shown in Figure 7.8 and Figure 7.9.

Using UniversalProcPtrs In Other Toolbox Calls

In this chapter you've seen that the universal header files define the `ModalFilterUPP` to be of type `UniversalProcPtr`:

```
typedef UniversalProcPtr ModalFilterUPP;
```

This is done for the purpose of adding clarity to your source code. Declaring a variable to be a `ModalFilterUPP` rather than a `UniversalProcPtr` makes it very evident as to the variable's purpose.



**FIGURE 7.8 A COMPILER THAT GENERATES POWERPC CODE
DEFINES A ModalFilterUPP AS A UniversalProcPtr.**

Once a `ModalFilterUPP` variable is declared, you use the `NewModalFilterProc()` routine to create a routine descriptor for the filter function:

```
ModalFilterUPP my_filter_UPP;

my_filter_UPP = NewModalFilterProc( My_Filter );
```

`ModalDialog()` isn't the only Toolbox routine that requires a `UniversalProcPtr`. Every Toolbox function that requires that one of its parameters be the address of a function requires a `UniversalProcPtr`. So the universal header files contain several typedefs that define `UniversalProcPtr` types that are descriptive of the purpose for which they'll be used.

Programming the PowerPC

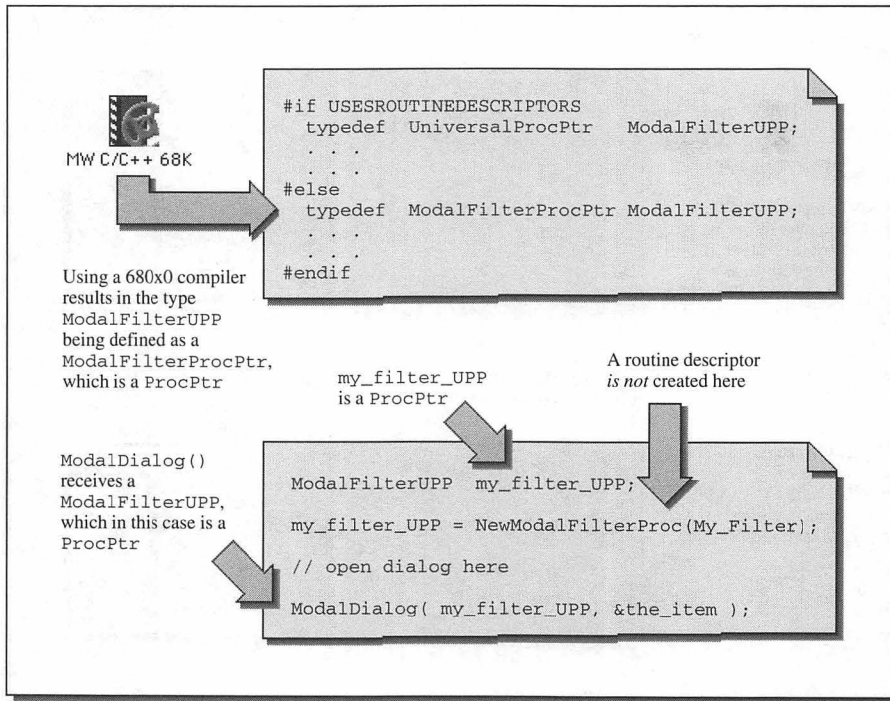


FIGURE 7.9 A COMPILER THAT GENERATES 680x0 CODE DEFINES A `ModalFilterUPP` AS A TYPE OF `ProcPtr`.

Consider `TrackControl()` as an example. This Toolbox routine accepts the address of an action procedure as its third parameter. The action procedure—which the programmer is responsible for writing—is called repeatedly while the user holds the mouse button down in a control. In the `Controls.h` universal header file you'll find this type definition:

```
typedef UniversalProcPtr ControlActionUPP;
```

Just as there is a function defined in the universal header files to create a filter function routine descriptor, so also is there a function defined to create an action procedure routine descriptor—`NewControlActionProc()`. Assuming you named your action routine `My_Action()`, your code would look similar to the following:

Chapter 7 Universal Procedure Pointers

```
ControlActionUPP my_action_UPP;  
  
my_action_UPP = NewControlActionProc( My_Action );
```



Passing the address of a function to Toolbox routines such as `ModalDialog()` and `TrackControl()` is an option. If you pass `nil` in place of a function address—as is often the case—you don’t need to declare a `UniversalProcPtr` or create a routine descriptor.

I’ve covered two of the most common `UniversalProcPtr`s, but there are a lot more. Here, from the universal header files, are several more:

```
typedef UniversalProcPtr ModalFilterUPP;  
typedef UniversalProcPtr ControlActionUPP;  
typedef UniversalProcPtr UserItemUPP;  
typedef UniversalProcPtr IOCompletionUPP;  
typedef UniversalProcPtr GrowZoneUPP;  
typedef UniversalProcPtr MenuDefUPP;  
typedef UniversalProcPtr MenuBarDefUPP;  
typedef UniversalProcPtr MCActionFilterUPP;  
typedef UniversalProcPtr DlgHookUPP;  
typedef UniversalProcPtr FileFilterUPP;  
typedef UniversalProcPtr DragGrayRgnUPP;  
typedef UniversalProcPtr WindowDefUPP;
```

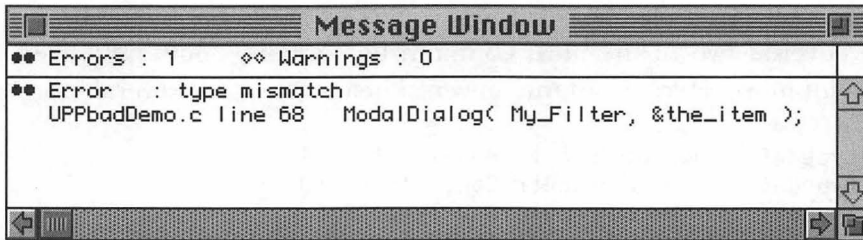


Remember, you don’t have to create a routine descriptor for every function in your program. You are only required to create a routine descriptor for routines that are called via a `ProcPtr`. That is, if the address of one of your functions is passed to a Toolbox routine, you must explicitly create a routine descriptor. If a routine in your program is directly called by another routine, it is not your responsibility to supply a routine descriptor or `UniversalProcPtr` for that routine.

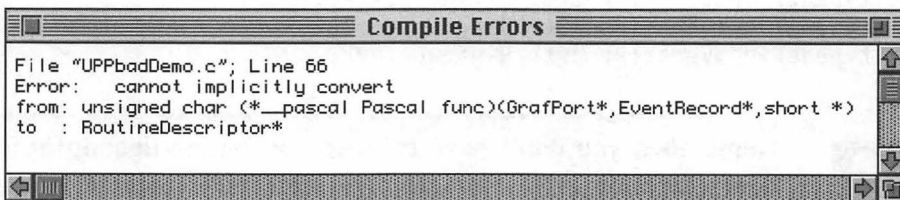
How will you know, or remember, just when to use one of the many `UniversalProcPtr`s? Keep in mind that they are only used for those few instances when you write a special routine whose address is passed to a

Programming the PowerPC

Toolbox function. You should quickly recognize those functions in your own code. And what of the times when you have to convert someone else's source code to PowerPC code? If you overlook something, don't worry—the compiler won't. When a PowerPC compiler looks for a `UniversalProcPtr` and doesn't find one, it will let you know. Figure 7.10 shows the error message you'll receive what compiling with the Metrowerks MW C/C++ PPC compiler. Figure 7.11 shows the error message generated by the Symantec CDK compiler.



**FIGURE 7.10 THE METROWERKS ERROR MESSAGE WHEN
A PROCPTR IS USED IN PLACE OF A UPP.**



**FIGURE 7.11 THE SYMANTEC ERROR MESSAGE WHEN
A PROCPTR IS USED IN PLACE OF A UPP.**

There is one instance, though, where a `UniversalProcPtr` is necessary, yet the compiler won't notice or report it if you forget to include one. The UPP type is a `UserItemUPP`, and the Toolbox call that uses the UPP is `SetDItem()`. `SetDItem()` can be used to change one of the properties of a dialog box item. When used for this purpose, the fourth argument to `SetDItem()` is a handle to the item. `SetDItem()` can also be used to

Chapter 7 Universal Procedure Pointers

associate a drawing function with a user item. When used in this manner, the fourth argument is a handle to the drawing routine. In either case, this fourth argument to `SetDItem()` is a handle—so the compiler can't insist that it be a `UniversalProcPtr`.



Later in this chapter you'll find the source code for a short program that includes a user item. Look for the program named `UPPdemo2`.

It will be your job to search your source code for any calls to `SetDItem()`. You'll also want to search for `SetDialogItem()`. The new universal header files now define both `SetDItem()` and the more descriptive `SetDialogItem()` to be one in the same routine—you can use either. If you come across a call to either routine, and it is used to associate a drawing routine with a user item, you'll need to modify the code to make use of a `UniversalProcPtr`. Assuming I wrote a user item drawing routine called `My_User()`, here's the 680x0 way of using `SetDItem()`:

```
// Open dialog, get item information

SetDItem( the_dialog, MAN_USER_ITEM, the_type,
          (Handle)My_User, &the_rect );
```

Compare that with the PowerPC way of doing things—declare a UPP, create a routine descriptor for the drawing routine, then typecast the UPP to a handle in the call to `SetDItem()`:

```
UserItemUPP my_user_UPP;

my_user_UPP = NewUserItemProc( My_User );

// Open dialog, get item information

SetDItem( the_dialog, MAN_USER_ITEM, the_type,
          (Handle)my_user_UPP, &the_rect );
```

Programming the PowerPC



As I mentioned, the PowerPC compiler will not notice if you forget to convert your `SetDItem()` calls to PowerPC code. Your code will successfully be turned into a standalone application. When you attempt to launch the program, however, it will crash.

UNIVERSALPROCPTR EXAMPLE PROGRAMS

The disk that accompanied this book has project files and source code files for a couple of programs that include `UniversalProcPtrs`. If you have CodeWarrior, look in the CodeWarrior Code *f* folder. If you own the Symantec Cross Development Kit (CDK), look inside the Symantec CDK Code *f* folder.

ModalDialog() and UPPs

Much of this chapter has used the `ModalDialog()` Toolbox function as an example of when to use a `UniversalProcPtr`. So it makes sense that the first complete UPP example involves the `ModalDialog()` function. This program, called `UPPdemo1`, should look familiar to you. It's the same as the example program listed in Chapter 6—with one addition. Here, `ModalDialog()` makes use of a filter function. And that means a UPP will have to be involved. Here's the source code listing in its entirety. An explanation follows.

```
//+++++ function prototypes ++++++

void    Initialize_Toolbox( void );
void    Open_Modal_Dialog( void );
pascal Boolean My_Filter( DialogPtr    dlog,
                          EventRecord *event,
                          short        *item );

//+++++ define directives ++++++
```

Chapter 7 Universal Procedure Pointers

```
#define    DIALOG_ID        128
#define    OK_BUTTON_ITEM    1

//+++++++ global variables ++++++

// If you're using the Symantec CDK, uncomment the
// following declaration. CodeWarrior users, leave as is

// QDGlobals  qd;

//+++++++ main ++++++

void main( void )
{
    Initialize_Toolbox();

    Open_Modal_Dialog();
}

//+++++++ initialize the Toolbox ++++++

void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}

//+++++++ open a modal dialog ++++++

void Open_Modal_Dialog( void )
{
    DialogPtr    the_dialog;
    short        the_item;
    Boolean      all_done = false;

    ModalFilterUPP  my_filter_UPP;
```

Programming the PowerPC

```
my_filter_UPP = NewModalFilterProc( My_Filter );

the_dialog = GetNewDialog( DIALOG_ID, nil,
                          (WindowPtr)-1L );
ShowWindow( the_dialog );

while ( all_done == false )
{
    ModalDialog( my_filter_UPP, &the_item );

    switch ( the_item )
    {
        case OK_BUTTON_ITEM:
            all_done = true;
            break;
    }
}
DisposeRoutineDescriptor( my_filter_UPP);
DisposDialog( the_dialog );
}

//+++++ filter function ++++++

pascal Boolean My_Filter( DialogPtr dlog,
                        EventRecord *event, short *item )
{
    long the_long;
    char chr;

    if ( event->what != keyDown )
        return ( false );

    chr = event->message & charCodeMask;

    if ( chr == 'x' )
    {
        *item = 1;
        return ( true );
    }

    return ( false );
}
```

The purpose of the ModalDialog() Toolbox function is to handle user actions in a dialog box. A filter function has that same purpose. It, how-

ever, handles events differently than `ModalDialog()` would. What events it handles, and how it handles them, is up to you. In the `UPPdemo1` program, the purpose of the filter function is to treat a press of the 'x' key the same as `ModalDialog()` would treat a mouse click on the OK button—the dialog box closes.

A filter function has three arguments. The first is a pointer to the dialog for which the filter will be used. The second is a pointer to the event record that holds the event which the filter will work with. The last argument is a pointer to a short. Here's the prototype for the filter function:

```
pascal Boolean My_Filter( DialogPtr   dlog,
                          EventRecord *event,
                          short       *item );
```

The function can assign this last argument the item number of an item in the dialog box. When the function completes, it will return a value of true if it handled the event and the `ModalDialog()` need do nothing more with the event. Returning a value of false tells `ModalDialog()` that while the filter may or may not have used or altered the event record, `ModalDialog()` should still process the event in its normal manner.

The filter function examines the event record to see if a key was pressed. If it wasn't, then `event->what` won't have a value of `keyDown`. In that case the filter function ends. A value of false is returned to let `ModalDialog()` know that the event wasn't processed—it should go ahead and do its thing.

```
if ( event->what != keyDown )
    return ( false );
```

If a key was pressed, the filter function carries on. It next determines which character was typed. If it was the letter 'x', the filter sets variable `item` to a value of 1—the item number of the OK button. It then returns a value of true to let `ModalDialog()` know that though the filter has altered things, further processing is needed by `ModalDialog()`. `ModalDialog()` will treat the event as it would if there was a mouse click in item number 1—it closes the dialog box.

Programming the PowerPC

```
chr = event->message & charCodeMask;

if ( chr == 'x' )
{
    *item = 1;
    return ( true );
}
```

If the program makes it to the end of the filter function, a value of false is returned. That tells `ModalDialog()` to go ahead and process the event as if nothing has happened.

UPPdemo1 is a PowerPC program. So the call to `ModalDialog()` needs to include a `UniversalProcPtr`. The `Open_Modal_Dialog()` routine first declares a `ModalFilterUPP` variable. It then calls `NewModalFilterProc()` to fill a routine descriptor with information about the `My_Filter()` filter function. `NewModalFilterProc()` also sets the `my_filter_UPP` to point to this new routine descriptor. When it comes time to call `ModalDialog()`, the `UniversalProcPtr` variable is passed.

```
ModalFilterUPP  my_filter_UPP;

my_filter_UPP = NewModalFilterProc( My_Filter );

// open dialog, then loop until done

ModalDialog( my_filter_UPP, &the_item );
```

Use your compiler to compile the UPPdemo1 code and to build a stand-alone application. If you have forgotten how to do that, refer back to Chapter 6. When you run UPPdemo1, note that you can click the mouse on the OK button or press the 'x' key to end the program.



Just a reminder that this is PowerPC code—so don't expect it to run on a 680x0-based Macintosh. In Chapter 8, you'll see how to get around this dilemma and produce applications that run on both the old and the new Macs.

Another Example of User Items and UPPs

If you want a picture to appear in a dialog box, you can include a picture item in the dialog's 'DITL' resource. As your standalone program runs, you can be assured that the picture will be properly updated as windows or alerts overlap it. That's because the Dialog Manager knows how to update 'DITL' items.

What about an instance where your program is to display a picture, but which picture is to be displayed isn't established until runtime? Perhaps the user gets to make a selection that determines which one of two pictures will show up in a dialog box. In a case such as this, you can't place a picture item in the 'DITL' of the resource file—you won't be able to fill in the resource ID of the 'PICT' that is associated with the picture item. True, you could simply draw the picture in the dialog box in response to the user's selection, but then it will be up to you to make sure the picture is properly updated every time the dialog becomes obscured and then comes back into view. A better solution is to make use of a user item.

The UPPdemo2 program opens a dialog box that displays a picture and two buttons. Clicking the mouse on the Post Alert button will cause an alert to open. The alert serves only as a test to see if the picture, which is partially obscured by the alert, gets properly redrawn when the alert is closed. Figure 7.12 shows what you'll see when you run UPPdemo2.

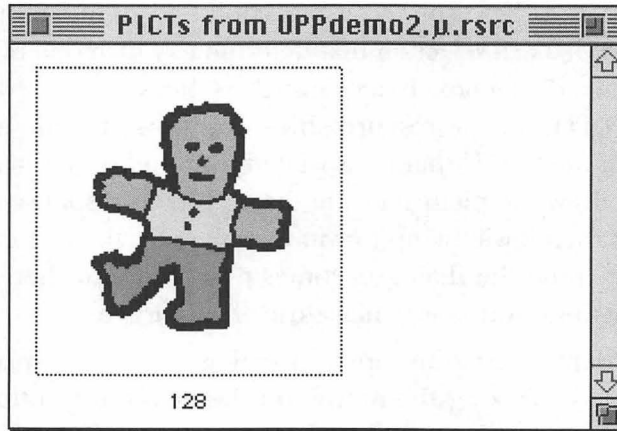


FIGURE 7.12 A LOOK AT THE UPPdemo2 PROGRAM.

Programming the PowerPC

When the alert is closed, the picture gets redrawn. The interesting part of the program isn't that the picture gets redrawn—you'd expect that in any Mac program. Rather, the point to note is that there is no update routine in the program. The picture gets redrawn without any effort on the part of the programmer.

The resource file for UPPdemo2 contains an 'ALRT' and a 'DITL' that holds the items in the alert. It also contains a 'PICT' to hold the picture that will go in the dialog box. Figure 7.13 shows that picture.

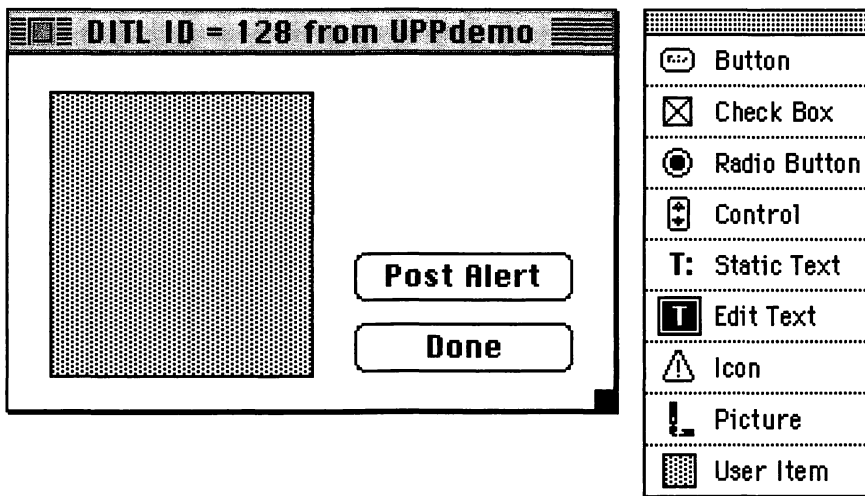


**FIGURE 7.13 THE SINGLE 'PICT' RESOURCE
IN THE RESOURCE FILE OF THE UPPdemo2 PROGRAM.**

Finally, the resource file holds the two resources that define the program's dialog box—a 'DLOG' and a 'DITL'. Figure 7.14 shows what the 'DITL' looks like. The gray rectangle is a user item. Though there's no connection between the user item and the 'PICT' resource in the resource file, there will be once the program runs.

The source code for UPPdemo2 appears below. You'll find an explanation immediately following it.

```
//+++++ function prototypes ++++++  
  
void    Initialize_Toolbox( void );
```



**FIGURE 7.14 THE SINGLE ‘DITL’ RESOURCE
IN THE RESOURCE FILE OF THE UPPdemo2 PROGRAM.**

```
void    Open_Dialog( void );
pascal void My_User( DialogPtr, short );

//+++++ define global constants +++++

#define    ALERT_ID                129
#define    DIALOG_ID              128
#define    DONE_BUTTON            1
#define    ALERT_BUTTON           2
#define    MAN_USER_ITEM          3
#define    THE_MAN_PICT_ID        128

//+++++ define global variables +++++

// If you're using the Symantec CDK, uncomment the
// following declaration. CodeWarrior users, leave as is

// QDGlobals  qd;

Boolean    All_Done = false;
EventRecord The_Event;
```

Programming the PowerPC

```
short          The_Picture_ID;

//+++++++ main listing ++++++

void main( void )
{
    Initialize_Toolbox();

    The_Picture_ID = THE_MAN_PICT_ID;

    Open_Dialog();

    while ( All_Done == false )
        ;
}

//+++++++ initialize the Toolbox ++++++

void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}

//+++++++ open a modal dialog ++++++

void Open_Dialog( void )
{
    short          the_type;
    Handle          the_handle;
    Rect            the_rect;
    DialogPtr       the_dialog;
    short           the_item;
    Boolean         dialog_done = false;

    UserItemUPP     my_user_UPP;

    my_user_UPP = NewUserItemProc( My_User );
}
```

Chapter 7 Universal Procedure Pointers

```
the_dialog = GetNewDialog( DIALOG_ID, nil, (WindowPtr)-1L );

GetDItem( the_dialog, MAN_USER_ITEM, &the_type,
          &the_handle, &the_rect );
SetDItem( the_dialog, MAN_USER_ITEM, the_type,
          (Handle)my_user_UPP, &the_rect );

ShowWindow( the_dialog );

while ( dialog_done == FALSE )
{
    ModalDialog( nil, &the_item );

    switch ( the_item )
    {
        case ALERT_BUTTON:
            Alert( ALERT_ID, nil );
            break;

        case DONE_BUTTON:
            dialog_done = true;
            All_Done = true;
            break;
    }
}
DisposeRoutineDescriptor( my_user_UPP);
DisposDialog( the_dialog );
}

//+++++++ user item drawing routine ++++++

pascal void My_User( DialogPtr the_dialog, short the_item )
{
    short    the_type;
    Handle    the_handle;
    Rect      user_rect;
    GrafPtr   old_port;
    PicHandle pict_handle;

    GetPort( &old_port );
    SetPort( the_dialog );

    GetDItem( the_dialog, the_item, &the_type,
              &the_handle, &user_rect );
    pict_handle = GetPicture( The_Picture_ID );
```

Programming the PowerPC

```
    DrawPicture( pict_handle, &user_rect );

    SetPort( old_port );
}
```

UPPdemo2 begins with the usual function prototypes and #define directives. All of the #defines are IDs of resources and 'DITL' items. The program contains three global variables: All_Done signals the end of the program, The_Event holds information about the most recent event, and The_Picture_ID contains the resource ID of the 'PICT' to display.

The program begins by initialing the Toolbox. Then, The_Picture_ID is set to the value of the 'PICT' resource. This assignment is made to demonstrate that the picture that is to be displayed in the dialog box could be determined at runtime. In a more involved program The_Picture_ID could be assigned its value in a statement such as the following:

```
switch ( users_picture_choice )
{
    case MAN_PICT:
        The_Picture_ID = THE_MAN_PICT_ID;
        break;
    case BOY_PICT:
        The_Picture_ID = THE_BOY_PICT_ID;
        break;
    default:
        The_Picture_ID = THE_DOG_COW_PICT_ID;
        break;
}
```

After assigning The_Picture_ID its value, the program opens a modal dialog in which to display the picture. Here's another look at main():

```
void main( void )
{
    Initialize_Toolbox();

    The_Picture_ID = THE_MAN_PICT_ID;

    Open_Dialog();
}
```

Chapter 7 Universal Procedure Pointers

```
        while ( All_Done == false )
        ;
    }
```

Open_Dialog() is the routine that holds the PowerPC code. This function declares a UserItemUPP variable named my_user_UPP. It then uses the NewUserItemProc() Toolbox function to create a routine descriptor for the My_User() drawing routine. The UPP variable my_user_UPP is assigned the address of this routine descriptor:

```
UserItemUPP  my_user_UPP;

my_user_UPP = NewUserItemProc( My_User );
```

Next, a call to GetNewDialog() opens the dialog box. GetDItem() is first called to get the dialog item information of interest—which dialog, which item, etc. Then SetDItem() is called. The information that was received from GetDItem() is passed to SetDItem(), along with the UniversalProcPtr. The fourth parameter passed to SetDItem() must always be a handle, so the UPP is typecast to one in the call.

```
the_dialog = GetNewDialog( DIALOG_ID, nil, (WindowPtr)-1L );

GetDItem( the_dialog, MAN_USER_ITEM, &the_type,
          &the_handle, &the_rect );
SetDItem( the_dialog, MAN_USER_ITEM, the_type,
          (Handle)my_user_UPP, &the_rect );
```

After the call to SetDItem(), the My_User() routine is bonded to the MAN_USER_ITEM. That means that whenever the MAN_USER_ITEM needs updating, the My_User() routine will be called. The system will take care of this call—it need not be explicitly made by your program. The remainder of the Open_Dialog() code is straightforward stuff. It displays the dialog box with a call to Show_Window(), repeatedly calls ModalDialog() to process user mouse clicks on dialog items, and closes the dialog box with a call to DisposeDialog().

The user item drawing routine accepts a pointer to the dialog box to draw to and the item number of the user item to draw into. The routine

Programming the PowerPC

saves the old port, then sets the port to that of the dialog. `GetDItem()` is called to get the size of the user item. `GetPicture()` is called to get a handle to the 'PICT' resource defined by `The_Picture_ID`. Then, `DrawPicture()` is called to draw the picture into the rectangle that was returned by `GetDItem()`. The routine ends by setting the port back to the original port.

```
pascal void My_User( DialogPtr the_dialog, short the_item )
{
    short      the_type;
    Handle     the_handle;
    Rect       user_rect;
    GrafPtr    old_port;
    PicHandle   pict_handle;

    GetPort( &old_port );
    SetPort( the_dialog );

    GetDItem( the_dialog, the_item, &the_type,
              &the_handle, &user_rect );
    pict_handle = GetPicture( The_Picture_ID );
    DrawPicture( pict_handle, &user_rect );

    SetPort( old_port );
}
```

CHAPTER SUMMARY

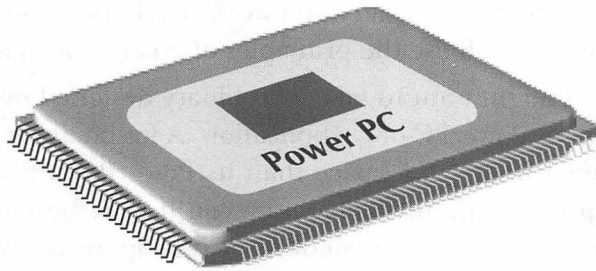
The Mixed Mode Manager allows a program to contain both 680x0 code and PowerPC code. The Mixed Mode Manager is responsible for keeping track of what mode the Mac should be in at all times—PowerPC or 68LC040 emulator. There are occasions, however, where the Mixed Mode Manager can't make this determination without an assist from the programmer. You'll do that by including a universal procedure pointer, or `UniversalProcPtr` in your code. This new pointer type enables you to pass the address of one of your functions to a Toolbox routine. In 680x0 development, this task was accomplished through the use of a `ProcPtr`.

A `UniversalProcPtr` is a pointer to a routine descriptor, or `RoutineDescriptor`. The `RoutineDescriptor` data structure holds

Chapter 7 Universal Procedure Pointers

information about a function, or routine, that is to be invoked. Among the many pieces of information in the `RoutineDescriptor` is a flag that specifies the instruction set architecture of the routine (680x0 or PowerPC), and a `ProcPtr` that, via the `TVector`, leads to the function's code.

On a Power Mac, to pass a pointer to a function you first declare a `UniversalProcPtr`. Next, you'll create a routine descriptor and set up the UPP using one of the Apple-implemented macros that were designed just for this reason. Lastly, you'll use the UPP as the parameter that is to serve as the pointer to the function.



CHAPTER 8

FAT BINARY APPLICATIONS

To give your program the broadest appeal, you'll want to ensure that it runs in fast, native mode on a Power Mac, yet is still compatible with the huge number of 680x0-based Macintoshes still on the market. To do that you'll turn your program into a *fat binary application*. Having a single program that runs on both 680x0-base Macs and PowerPC-based Macs means you won't have to supply the user with two separate versions of your program.

There may be occasions when you know a program will be run only on a Power Macintosh. Perhaps the program was designed to take advantage of the speed of the PowerPC, and won't run as smoothly on a 680x0-based Mac. Or, you may have custom written the application for someone who has a Power Mac—or a network of Power Macs. For applications like this, you'll want to keep your program as a PowerPC-only application. You will, however, want to let users know why their attempts to run the

Programming the PowerPC

program on a 680x0-based Mac won't work. To do this, you won't create a fat binary—but you will use the principles of creating a fat application.

Finally, you might want to take a fat binary designed by someone else and turn it into a PowerPC-only application. A fat binary is large in size—it takes up much more disk space than its PowerPC-only version. If you have a fat application, and you know you'll only be running it on a PowerPC, you can reduce the amount of disk space it occupies by converting it to a PowerPC application.

FAT APPLICATION THEORY

Turning your program into a fat binary application makes it backward compatible. That is, it will be compatible with pre-Power Macintosh era machines. In this section I'll cover the generalities of fat binaries. Much of the rest of the chapter will be devoted to specific instances of creating fat binaries.

Applications and 680x0/PowerPC Compatibility

A Mac application created using a compiler designed to generate code for a 680x0 Mac will be able to run on any Macintosh—including Power Macs. However, when run on a Power Macintosh the application will be running in emulation mode. So while a 680x0, or 68K, program will run on a Power Mac, it won't be taking advantage of the speed gains of the PowerPC processor. Figure 8.1 uses the Metrowerks 68K compiler icon in an illustration of this idea.

The apparent solution is to use a PowerPC cross-compiler to generate code that runs in the native mode on a Power Mac. This is only a partial solution, though. An application built to run on a Power Mac does so exclusively—it won't run on a 680x0 Mac. This is because PowerPC instructions will not be recognized by the 680x0 processor. In Figure 8.2 the icon for Metrowerks other C/C++ compiler—a PowerPC version—is used to illustrate this.

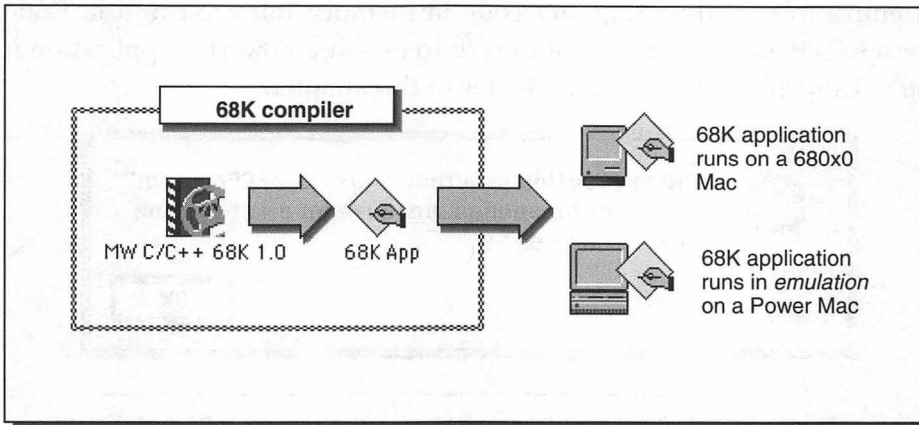


FIGURE 8.1 A 68K COMPILER GENERATES EXECUTABLES THAT RUN ON BOTH 68K AND POWERPC SYSTEMS.

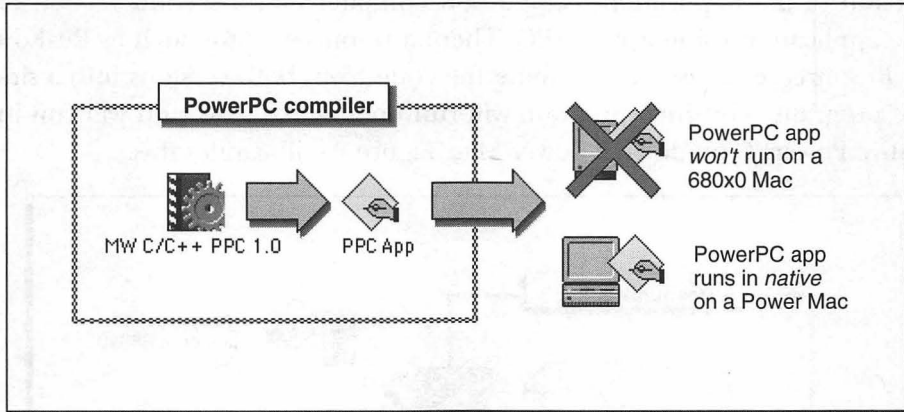


FIGURE 8.2 A POWERPC COMPILER GENERATES EXECUTABLES THAT RUN ON ONLY POWERPC SYSTEMS.

If you create a PowerPC version of a program and attempt to run it from the desktop of a 680x0 Macintosh, you'll encounter the system alert shown in Figure 8.3. An error ID of -192 represents a "resource not found" error. This error comes about when the Process Manager

Programming the PowerPC

attempts to load the program's code in memory, but can't find it. Code for a 680x0 application is kept in resources—in a PowerPC application it isn't. This will be fully explained later in this chapter.

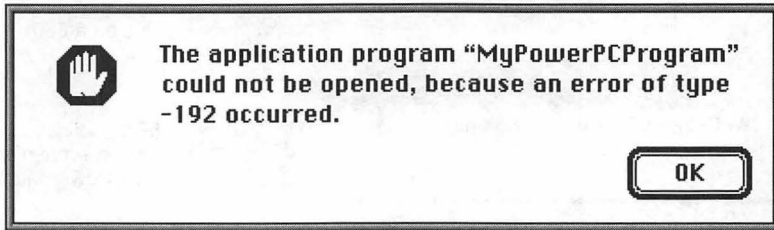


FIGURE 8.3 A POWERPC APPLICATION WILL NOT RUN ON A 68K MAC.

The complete solution is to build a *fat application*—also referred to as a *fat app* or *fat binary*. A fat application contains all of the compiled code for two separate versions of a single application. This is achieved by building one version of the application using a 68K compiler and a second version of the application using a PowerPC. Then, a resource editor such as ResEdit or Resourcerer is used to combine the code from both versions into a single program. This final program will run on a 680x0 Mac, and will run in native PowerPC mode on a Power Mac. Figure 8.4 illustrates this.

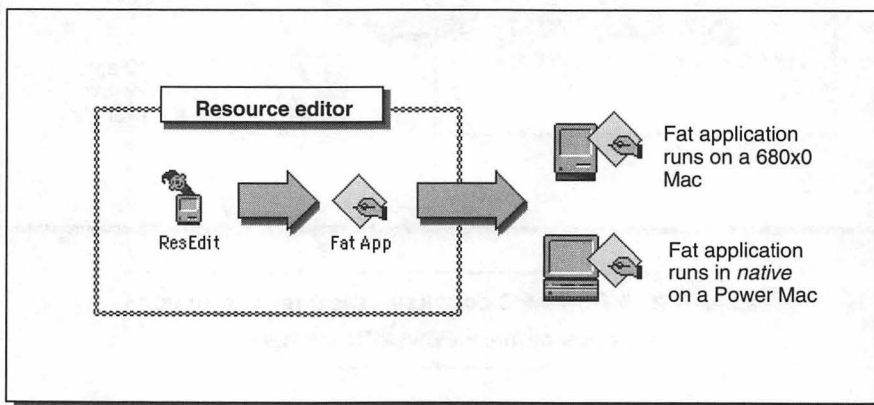


FIGURE 8.4 A FAT BINARY IS CREATED BY COMBINING THE RESOURCES OF TWO VERSIONS OF THE SAME PROGRAM.

Two separate versions of a program are created, then, from these two versions, a single fat application is formed. Figure 8.5 summarizes the process of making a fat application.

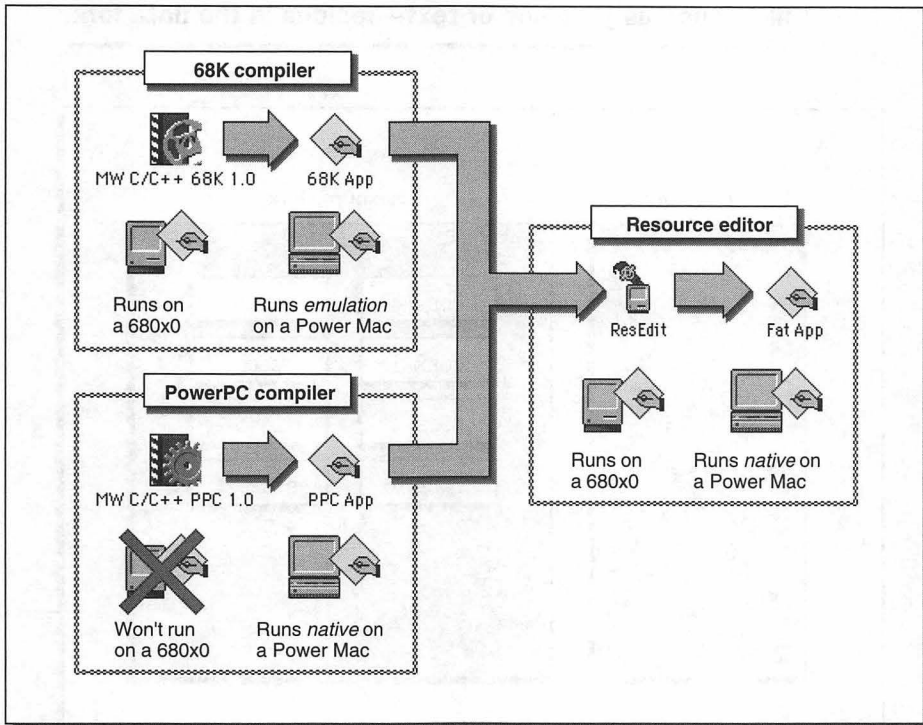


FIGURE 8.5 DIFFERENT VERSIONS OF THE SAME PROGRAM RUN ON DIFFERENT SYSTEMS.

Structure of a 680x0 Application

All Macintosh files—including applications—can contain two *forks*: a resource fork and a data fork. A fork can be empty, or even nonexistent for any one file. For a 680x0 application file, both the code that makes up the application and the resources that the application uses are stored in the resource fork. The data fork will usually be empty. Figure 8.6 shows the structure of a 680x0 application.



The structure of a document file created by an application is generally the opposite of that of an application. In a document, the resource fork is rarely used. The bulk of the file—such as graphics or text—resides in the data fork.

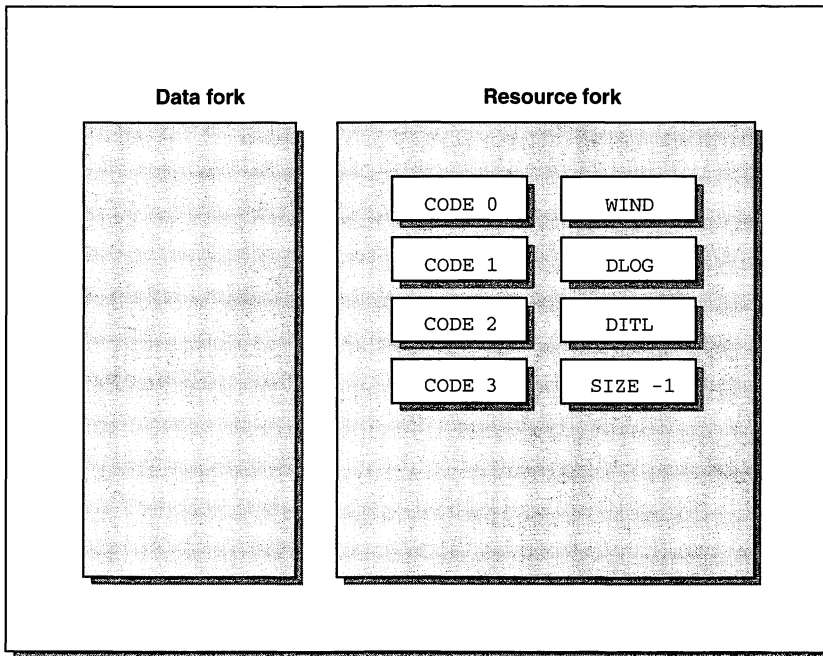


FIGURE 8.6 THE DATA FORK OF A 680x0 APPLICATION IS USUALLY EMPTY.

Figure 8.6 shows that the executable code that makes up a 680x0 application is stored in one or more 'CODE' resources. Since no one 'CODE' resource is allowed to exceed 32K in size, most programs contain more than one 'CODE' resource—the figure arbitrarily shows four 'CODE' resources. When a user double-clicks on a 680x0 application icon, the Segment Manager looks to the application's resource fork. It loads the application's code from the 'CODE' resources that are found there.

Figure 8.6 also shows that the resources that are typically found in all Mac programs, such as ‘DLOG’ and ‘DITL’ resources, reside in the resource fork. With the arrival of System 7 Apple requested that developers include a ‘SIZE’ resource with an ID of –1 in each application. The ‘SIZE’ resource contains information that the Process Manager needs in order to launch and allocate memory for an application. Figure 8.6 includes a ‘SIZE’ resource in the resource fork.

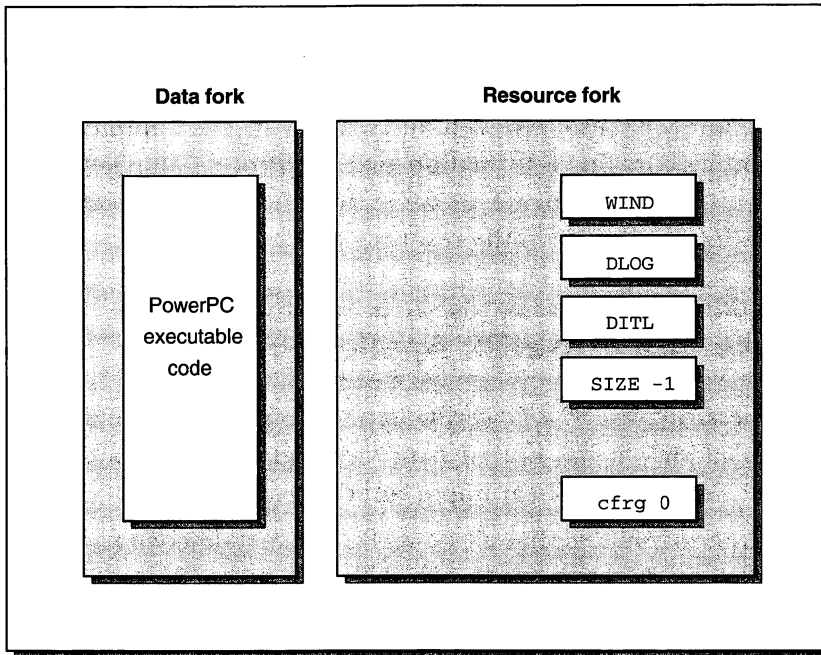
Structure of a PowerPC Application

The general structure of a PowerPC application is the same as that of a 680x0 application—it has both a data fork and resource fork. The contents of the two, however, bear important differences.

In Chapter 5 you saw that all PowerPC executable code—whether for an application, code resource, or shared library—is considered a code fragment. To let the Process Manager know that it is dealing with a PowerPC code fragment rather than a 680x0 application, the resource fork of a PowerPC application contains a type of resource not found in 680x0 applications—the ‘cfrg’ resource. Besides telling the Process Manager that the application is a PowerPC application, the ‘cfrg’ resource—which has an ID of 0—indicates where the executable code is located. Unlike a 680x0 application, the executable code of the PowerPC application is usually found in the application’s data fork. Figure 8.7 illustrates this.

Structure of a Fat Application

When an application launches on a 680x0-based Mac, the Process Manager looks for a ‘CODE’ resource with ID 0 in the applications resource fork. If it finds one, it loads that ‘CODE’ resource and any other ‘CODE’ resources that are marked for preloading. If executable code exists in the application’s data fork, it will be ignored. Likewise, if a ‘cfrg’ resource exists it too will be ignored—the ‘cfrg’ resource type didn’t exist prior to the PowerPC-based Macs.



**FIGURE 8.7 THE DATA FORK OF A POWERPC APPLICATION
HOLDS THE APPLICATION'S EXECUTABLE CODE.**

When an application launches on a PowerPC-based Mac, the Process Manager first looks for and examines the 'cfrg' resource with ID 0. This resource indicates where the executable code for the application is located—usually in the application's data fork. If 'CODE' resources are present in the resource fork, the Process Manager will simply ignore them.

From the above two paragraphs it should be evident that if a single application contains executable code in two separate but complete formats—one version in the data fork and one version as 'CODE' resources in the resource fork—than that application will be able to run on either a 680x0-based Macintosh or a PowerPC-based Mac. That is, in fact, just how a fat binary application is set up. Figure 8.8 illustrates the structure of a fat app.

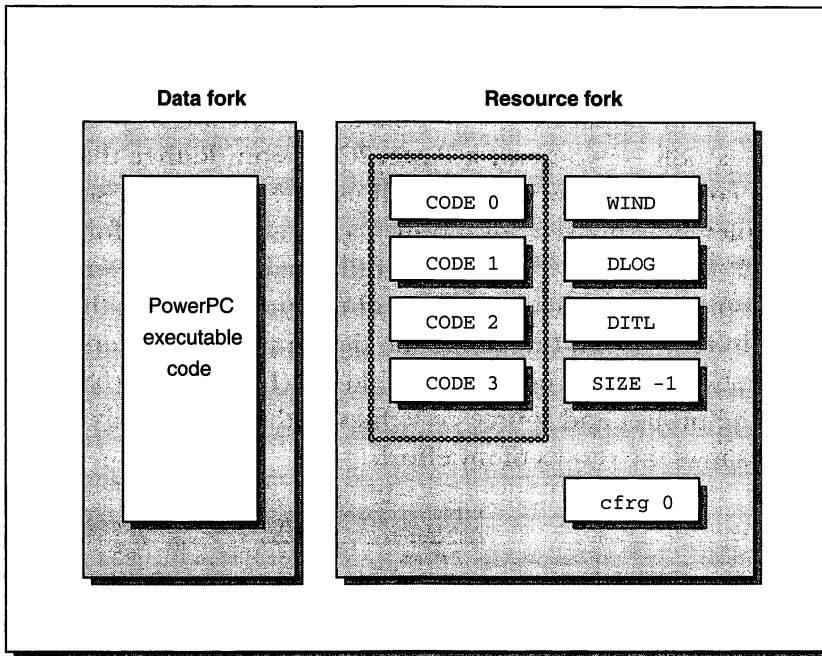


FIGURE 8.8 A FAT BINARY HOLDS TWO VERSIONS OF AN APPLICATION'S EXECUTABLE CODE.

Not only will the fat application run on either type of Macintosh, but it will run native on the PowerPC. Without the executable code in the data fork and without the 'cfrg' resource, the application would still run on a PowerPC—but it would be the old, nonported 680x0 code that would run—not the faster, PowerPC code.

Creating a fat application requires that you compile the same code twice—once using a 680x0 compiler and once using a PowerPC compiler. Then, using a resource editor the resulting applications are merged together to form a single fat application. Exactly how that is done is the topic of the next two sections. If your development system is Symantec's CDK, you might want to skip the next section and move right on to the *Using Symantec's CDK to Create Fat Apps* section. Or, if you haven't yet decided on a development system you might want to read both of the next two sections to see which system looks more appealing.

USING CODEWARRIOR TO CREATE FAT APPS

To create a fat application you'll make two versions of the same program—a 68K version and a PowerPC version. Rather than go over the source code for a new program, I'll use the UPPdemo1 source code from Chapter 7 as my starting point. I created two new folders—one named UPP Demo 1 (PPC) *f* and the other named UPP Demo 1 (68K) *f*. I then copied the source code file and resource file from the (7) UPP Demo 1 *f* to each. Next, I renamed the files in the new folders. Since I'll be creating two similar programs, I included "(PPC)" and "(68K)" in the names of the project and source code files so that I won't mix things up. Figure 8.9 shows the results of my effort.

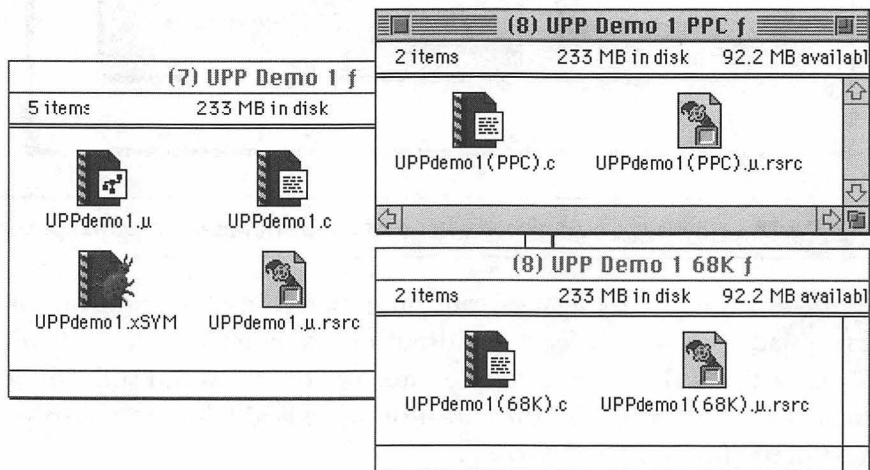


FIGURE 8.9 CREATE TWO FOLDERS, ONE TO HOLD A 68K PROJECT AND ONE TO HOLD A POWERPC PROJECT.

Creating the PowerPC Version

To create the fat application you'll use both of CodeWarrior's two C/C++ compilers. Begin by running the MW C/C++ PPC compiler.

Chapter 8 Fat Binary Applications

Create a new project named `UPPdemo1(PPC).µ`, making sure to save it in the `UPP Demo 1 (PPC) f` folder. Then add the source code file and the three libraries that are common to all PowerPC projects. Figure 8.10 shows the project window for my project. The source code file is a copy of the first `UniversalProcPtr` example found in Chapter 7. If you'd like to see the source code, refer back to the `UPPdemo1` example in Chapter 7.

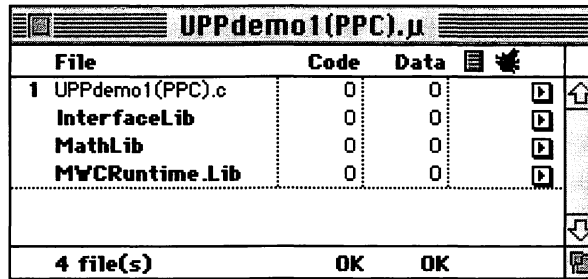


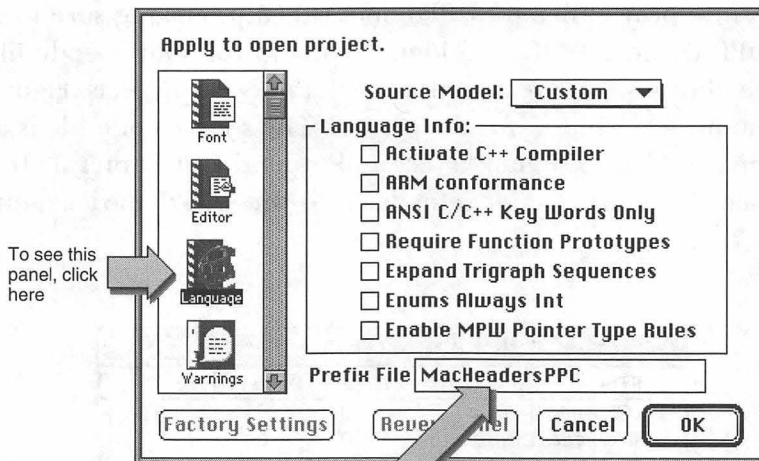
FIGURE 8.10 THE PROJECT WINDOW FOR THE POWERPC PROJECT.

To verify that the correct prefix file is being used, select **Preferences** from the **Edit** menu. When you do you'll see the Preferences dialog box. Click the **Language** icon to view the message area that includes the Prefix File edit box. For compiling with the PPC compiler, the prefix file must be the `MacHeadersPPC` file—as shown in Figure 8.11.

While you're at the Preferences dialog box, click on the **Project** icon. That displays a new pane of information in the dialog box. Type in a file-name—as shown in Figure 8.12. Then click the **OK** button.

After dismissing the Preference dialog box, select **Make** from the **Project** menu. That creates a PowerPC version of the program. Select **Quit** from the **Edit** menu to return to the desktop.

Programming the PowerPC



Make sure the MacHeadersPPC file is listed here when running the PowerPC version of CodeWarrior

FIGURE 8.11 ALL CODEWARRIOR POWERPC PROJECTS USE THE MACHEADERSPPC PREFIX FILE.

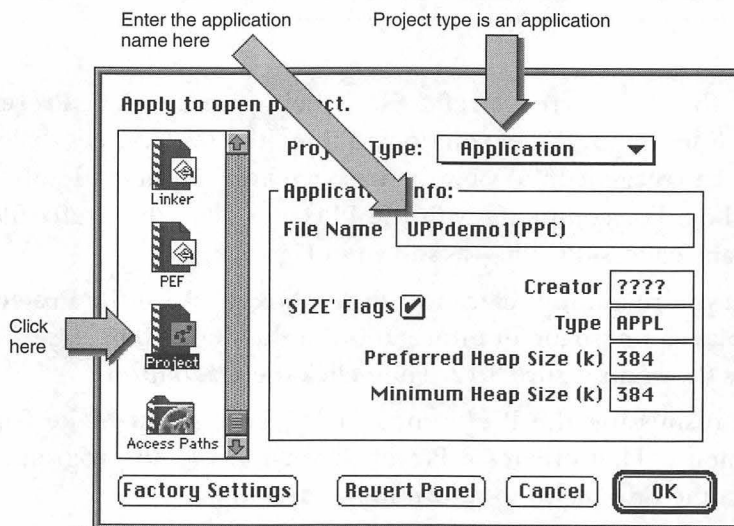


FIGURE 8.12 SET THE POWERPC PROGRAM'S NAME BEFORE BUILDING THE APPLICATION.

Creating the 680x0 Version

Now it's time to create a second version of the UPPdemo1 application. This time launch the 68K version of CodeWarrior—the MW C/C++ 68K compiler. Select **New** from the File menu to create a new project. Save it in the UPP Demo 1 (68K) *f* folder as UPPdemo1(68K).μ. Add the source code file and the one library that's required for all 68K projects—MacOS.lib. Figure 8.13 shows what my project window looks like.

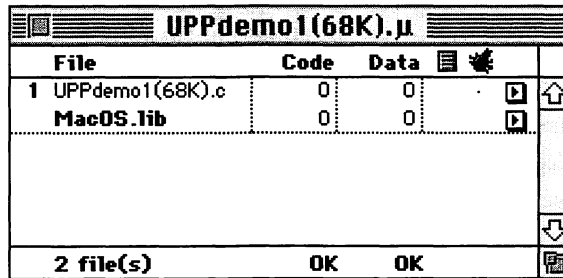


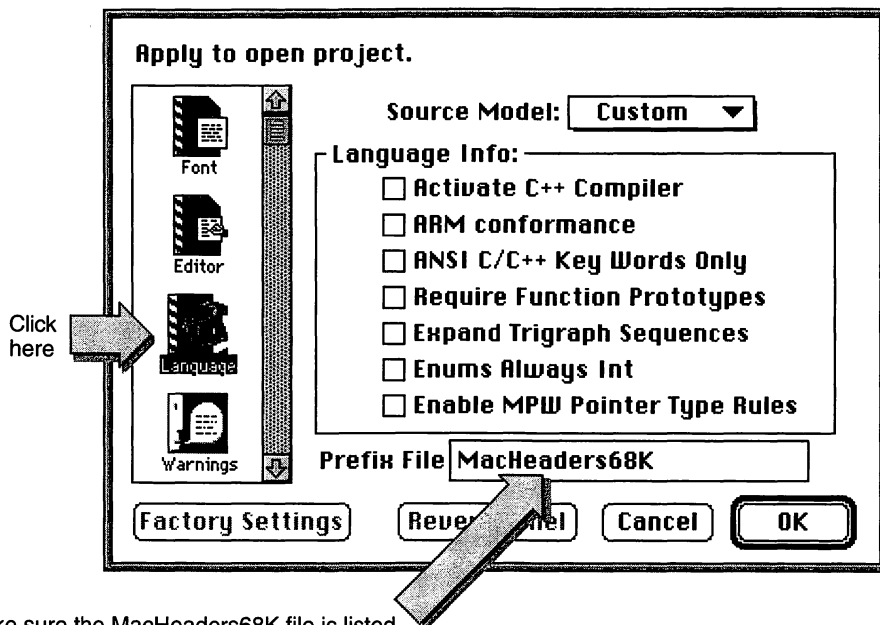
FIGURE 8.13 THE PROJECT WINDOW FOR THE 68K PROJECT.

Recall that I copied the original UPPdemo1.c file from my Chapter 7 example—an example written to compile using a PowerPC compiler. With that in mind, do I now have to change any of the source code in order to compile it using the 68K compiler? No. The same source code can be used for both the 68K and PPC compilers. That's because CodeWarrior makes use of Apple's universal header files. Once I have my source code able to compile using a compiler that generates PowerPC executables, I can also use the same source code with a compiler that generates 68K executables.

In Chapter 7 you read that the two Metrowerks C/C++ compilers require different MacHeaders prefix files. When compiling with the CodeWarrior PPC compiler, the MacHeadersPPC file must be the prefix file. For the CodeWarrior 68K compiler, the prefix file is MacHeaders68K. To verify that the 68K compiler is using the proper prefix file, select **Preferences** from the Edit menu. Click the **Language** icon

Programming the PowerPC

in the Preferences dialog box to have the dialog display the proper panel. Make sure that the MacHeaders68K file is named in this edit box. If it's not, type it in—as shown in Figure 8.14.



Make sure the MacHeaders68K file is listed here when running the 68K version of CodeWarrior

**FIGURE 8.14 ALL CODEWARRIOR 68K PROJECTS
USE THE MACHEADERS68K PREFIX FILE.**

Before dismissing the Preferences dialog box, click the **Project** icon. Type in a program name that will distinguish this version of UPPdemo1 from the PowerPC version—as I've done in Figure 8.15.

Click the **OK** button to dismiss the Preferences dialog box. Then select **Make** from the Project menu. In a matter of seconds you'll have a 68K version of the UPPdemo1 program. Select **Quit** from the File menu to exit the CodeWarrior compiler. When you do, you'll find that you have a second version of the UPPdemo1 program.

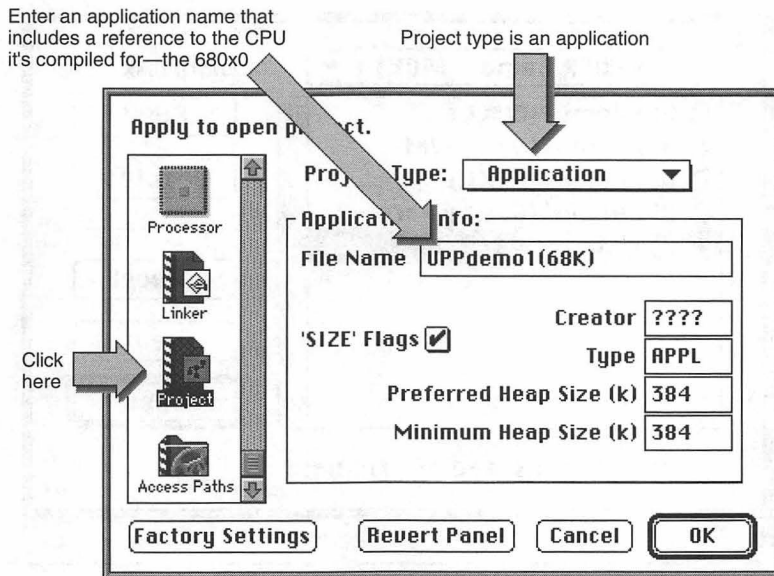


FIGURE 8.15 SET THE 68K PROGRAM'S NAME BEFORE BUILDING THE APPLICATION.

Creating the Fat Binary

To create a fat binary version of the UPPdemo1 program you'll need to use a resource editor. Launch the editor and work your way into the UPP Demo 1 (68K) folder as I'm doing in Figure 8.16. Open the UPPdemo1(68K) program (*not* the resource file).



You'll be turning the PowerPC version of the program into a fat application. If for some reason you would like to retain a "PowerPC-only" application, make a copy of the UPPdemo1 (PPC) program before altering it with the resource editor.

Next, open the PowerPC version of the program. Figure 8.17 shows the resource files for both applications. Note that there is no 'CODE' resources in the PowerPC version, and no 'cfrg' resource in the 68K version—just as expected.

Programming the PowerPC

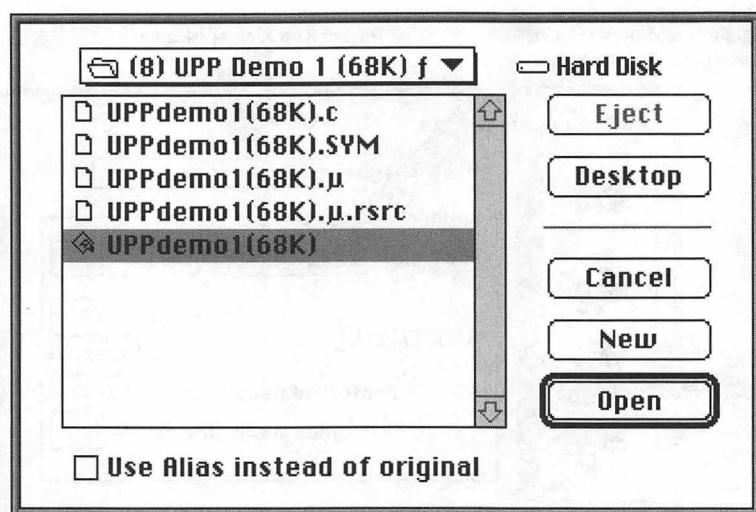


FIGURE 8.16 OPENING THE 68K APPLICATION FROM THE RESOURCE EDITOR.

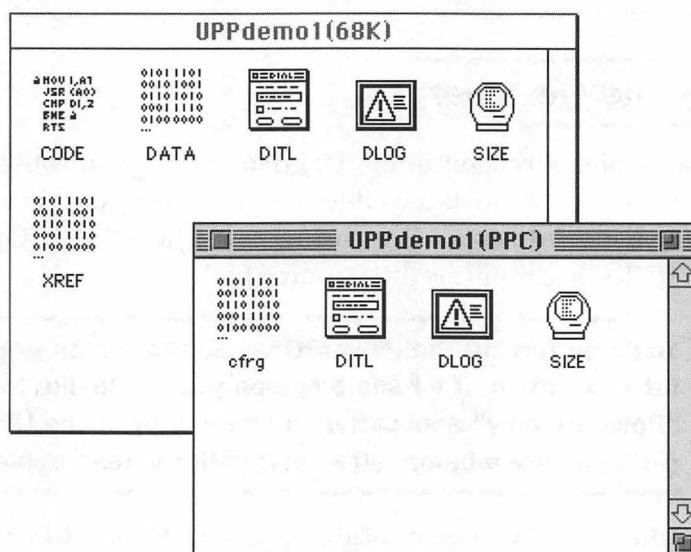


FIGURE 8.17 THE RESOURCE FILES FOR BOTH VERSIONS OF THE UPPdemo1 PROGRAM.

Now click on the CODE icon in the 68K resource file to select the ‘CODE’ resources. Select Copy from the Edit menu. Then click on the PowerPC resource file and select Paste from the Edit menu. Do the same with each of the resources that appear in the 68K version but not in the PowerPC version. That includes the ‘CODE,’ ‘DATA,’ and ‘XREF’ resources. When you’re done, your resource files should look like the ones pictured in Figure 8.18.

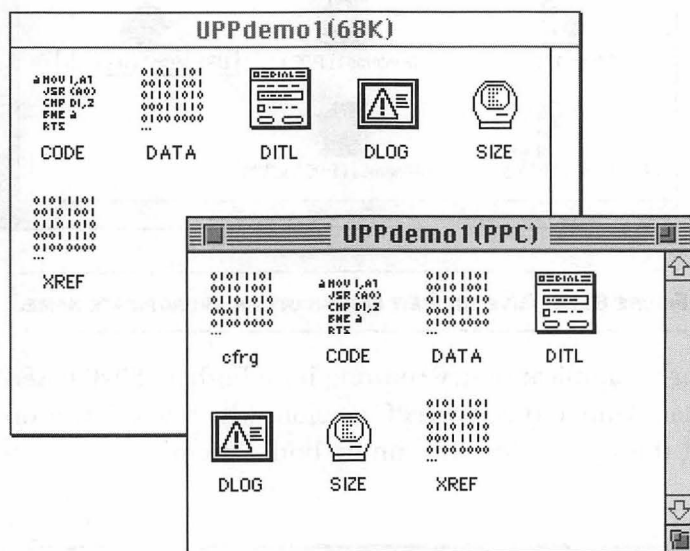


FIGURE 8.18 THE RESOURCE FILES AFTER THE 68K RESOURCES HAVE BEEN COPIED TO THE POWERPC RESOURCE FILE.

You’ve just created a fat binary application. The PowerPC version now contains all its original resources as well as the resources that were unique to the 68K version of the program. Save the resource files and exit the resource editor.

To make it obvious that the PowerPC version of the program is now a fat binary, rename it to something like UPPdemo1(fat app)—as I’m doing in Figure 8.19.

Programming the PowerPC

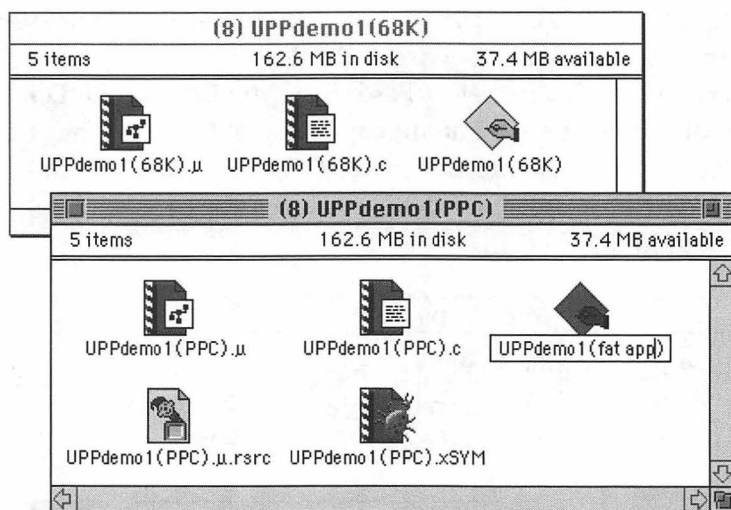


FIGURE 8.19 GIVE THE NEW FAT BINARY AN APPROPRIATE NAME.

To test your fat application try running it on both a 680x0-based Mac and a Power Mac. Unlike the PowerPC version, which won't run on a 680x0-based Mac, this application will run on both systems.

USING SYMANTEC'S CDK TO CREATE FAT APPS

A fat application starts out as two versions of the same program—a 68K version and a PowerPC version. To save a little typing, I'll use the UPPdemo1 source code from Chapter 7 and turn it into a fat binary. I begin by making two new folders—one named UPP Demo 1 (PPC) *f* and the other named UPP Demo 1 (68K) *f*. Next, I copy the six files that I need for any Symantec CDK project to the PowerPC folder. Then I copy the source code file and resource file from the (7) UPP Demo 1 *f* to the 68K folder—Symantec projects that result in 680x0 applications don't need the extra PowerPC files. Finally, I rename the copied files, making sure to include “(PPC)” and “(68K)” in the names so that I can keep track of things. Figure 8.20 shows the original folder and the two new ones.

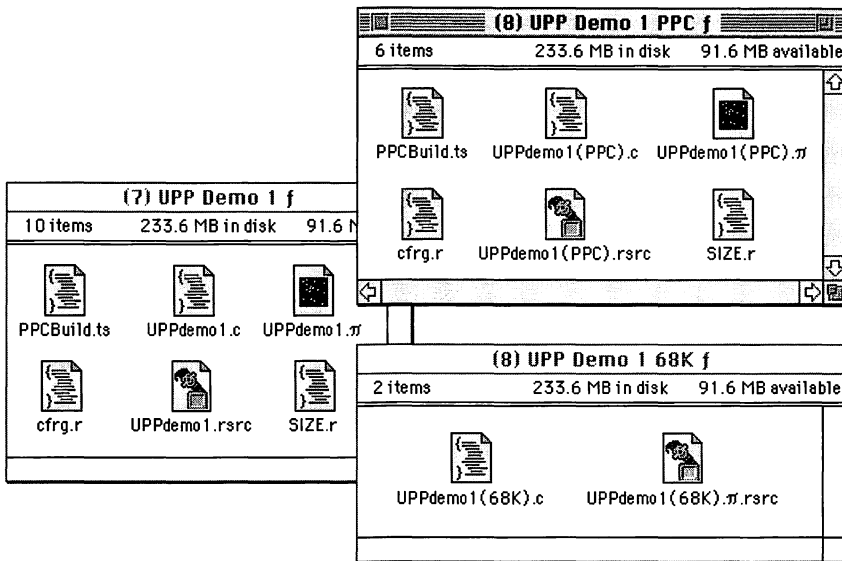


FIGURE 8.20 CREATE TWO FOLDERS, ONE TO HOLD A 68K PROJECT AND ONE TO HOLD A POWERPC PROJECT.

Creating the PowerPC Version

Begin by double-clicking on the PowerPC project to launch the THINK Project Manager. The project window that opens will hold just about all the files you need for the project. Add the UPPdemo1(PPC).c file and the UPPdemo(PPC).rsrc file using the **Add Files** menu item. Then remove the UPPdemo1.c and UPPdemo1.rsrc files—they're remnants of the original UPPdemo1 project from Chapter 7. Figure 8.21 shows what your project window should look like.

Double-click on the cfrg.r file to open it. This file contains the name that the standalone application will be given. Type in a new name, such as UPPdemo1(PPC). That's what I've done in Figure 8.22.

Now you're all set to create the PowerPC version of the program. Select **Remove Objects** from the Project menu, then choose **Bring Up To Date** from the same menu. Next, select **Build PowerPC App** from the

Programming the PowerPC

AppleScript menu. After a few moments the build will be complete and you'll have one of the two applications you need.

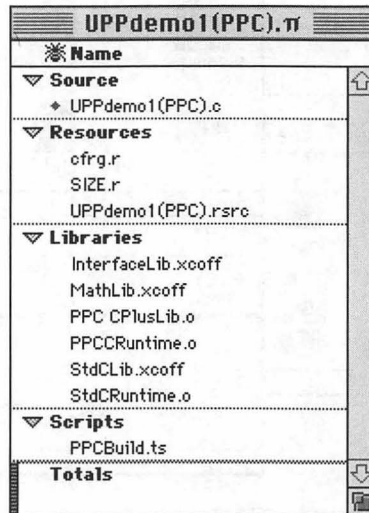


FIGURE 8.21 THE PROJECT WINDOW FOR THE POWERPC PROJECT.

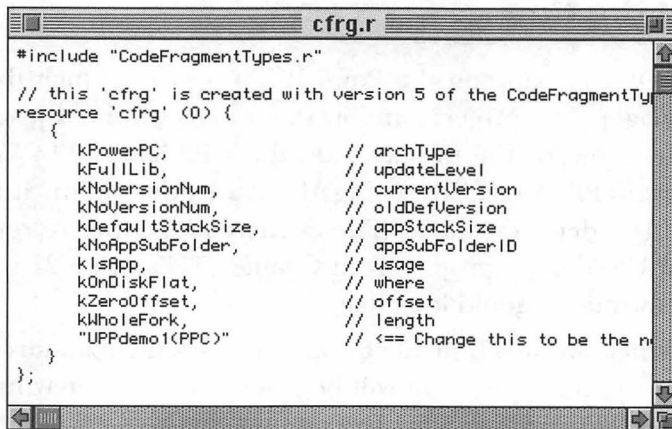


FIGURE 8.22 EDIT THE CFRG.R FILE TO SET THE APPLICATION'S NAME.

Creating the 680x0 Version

Now you need to make a second version of the UPPdemo1 application. If you've left the THINK environment, restart it now. If you're still in the THINK environment, close the PowerPC project now. The New Project dialog box will open. Click once on the words **Empty Project** in the list, and click once in the **Create folder** checkbox to uncheck it. The New Project dialog box is shown in Figure 8.23.

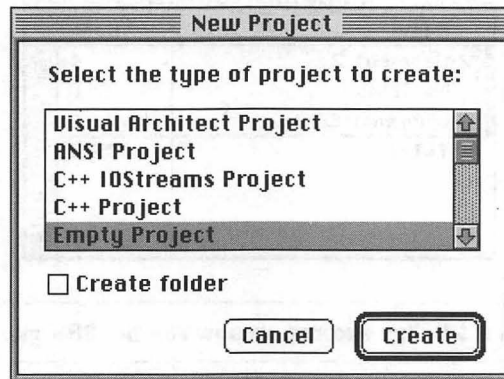


FIGURE 8.23 SELECT THE EMPTY PROJECT OPTION AND UNCHECK THE CREATE FOLDER BOX WHEN CREATING A NEW PROJECT.

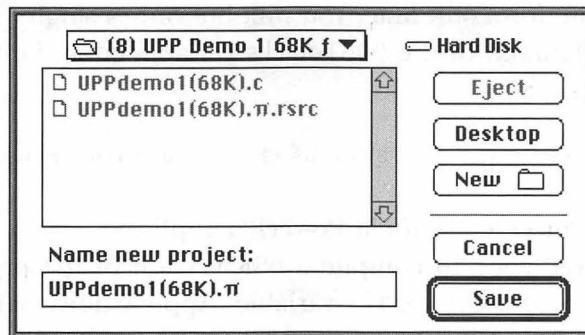
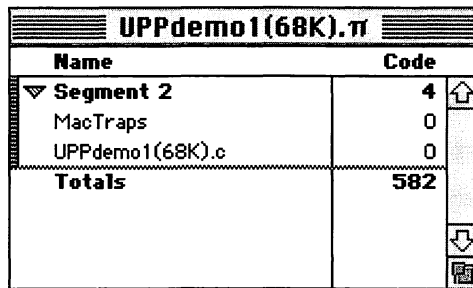


FIGURE 8.24 NAMING AND SAVING THE NEW 68K PROJECT.

Programming the PowerPC

Click the **Create** button. The dialog box will close and another one will open—it's shown in Figure 8.24. After working your way into the UPP Demo 1 68K *f* folder, enter the name UPPdemo1(68K).*π* for the project. Then click the Save button.

An empty project window will open. Add the source code file and the MacTraps library to the project. Your project window should then look like the one pictured in Figure 8.25.



Name	Code
▼ Segment 2	4
MacTraps	0
UPPdemo1(68K).c	0
Totals	582

FIGURE 8.25 THE PROJECT WINDOW FOR THE 68K PROJECT.

The UPPdemo1(68K).c source code file is a copy of the UPPdemo1.c file from Chapter 7. This source code was written to compile using a PowerPC compiler and run on a Power Mac. The question now arises as to whether or not you have to change any of the source code in order to get it to compile for a 68K Mac. You will, but only a single line. Look for the global declaration of the QDGlobals variable *qd* and either delete it or comment it out:

```
// QDGlobals qd;  <- only used for building PowerPC apps
```

After writing source code for a PowerPC application, you can use that very same source code to compile a 68K version of the program—with the exception of the QDGlobals variable. Apple's universal header files make this possible.

To create the 68K version of the program, select **Build Application** from the Project menu. That menu is shown in Figure 8.26. Note that

you don't use a menu option from the AppleScript menu. The AppleScript menu is only used by the Cross Development Kit to build PowerPC applications.



FIGURE 8.26 THE SYMANTEC PROJECT MENU.

When the dialog box pictured in Figure 8.27 prompts you for an application name, enter `UPPdemo1(68K)`. That will distinguish this application from your PowerPC version.

Click the **Save** button. The Symantec compiler will compile and link the code to build an application. If you forgot to comment out the variable `qd`, you'll see the error message displayed in Figure 8.28. Open the source code file and comment out the single line of code that declares the `qd` variable. Then select **Build Application** again.

After successfully building the application, select **Quit** from the File menu to exit the THINK Project Manager. You'll now have two versions of the `UPPdemo1` program.

Programming the PowerPC

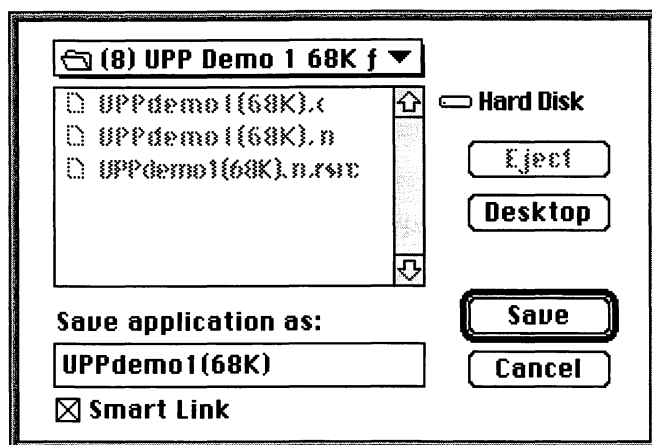


FIGURE 8.27 NAMING THE 68K APPLICATION.

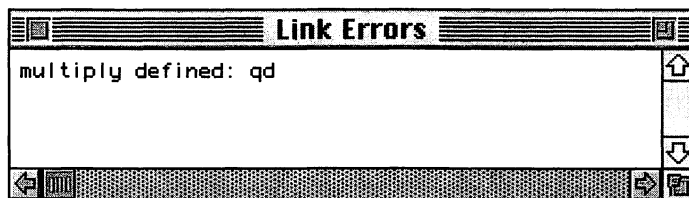


FIGURE 8.28 FORGETTING TO COMMENT OUT THE QD VARIABLE DECLARATION IN THE 68K VERSION RESULTS IN A LINK ERROR.

Creating the Fat Binary

Creating a fat binary version of the UPPdemo1 program entails the use of a resource editor. I'll be using ResEdit for the following example. Run the editor and work your way into the UPP Demo 1 (68K) *f* folder, as shown in Figure 8.29.



NOTE

You'll be copying resources from the 68K version and pasting them into the PowerPC version—turning the PowerPC version into a fat application. If for some reason you want

to retain a “PowerPC-only” application, make a copy of the UPPdemo1 (PPC) program before editing it.

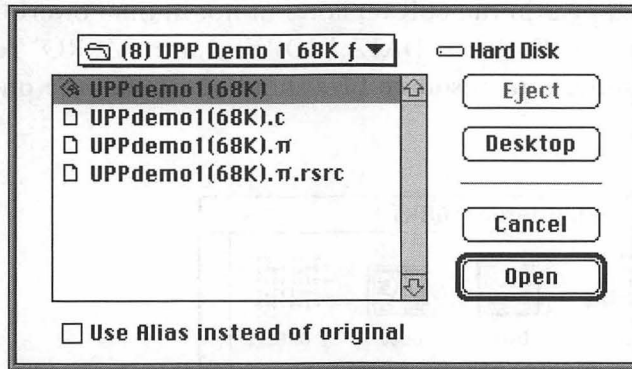


FIGURE 8.29 OPENING THE 68K APPLICATION FROM THE RESOURCE EDITOR.

Now open the PowerPC version of the program. Figure 8.30 shows the resource files for both versions of the program. Note that the PowerPC version contains no ‘CODE’ resources, and the 68K version doesn’t have a ‘cfrg’ resource—as expected.

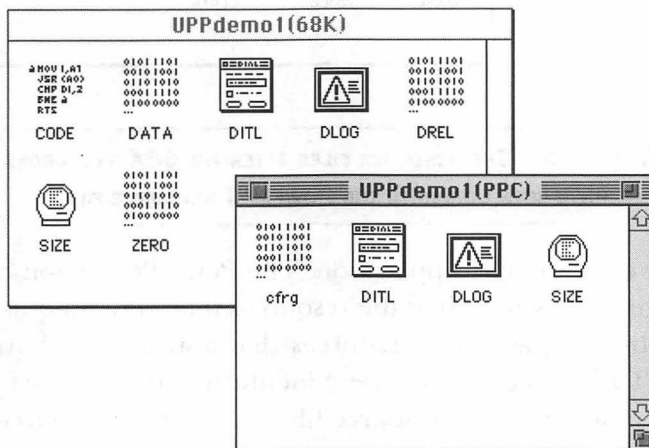


FIGURE 8.30 THE RESOURCE FILES FOR BOTH VERSIONS OF THE UPPdemo1 PROGRAM.

Click on the **CODE** icon in the 68K resource file to select the ‘CODE’ resources. Select **Copy** from the Edit menu. Click on the PowerPC resource file and select **Paste** from the Edit menu. Do this for each of the resources that appear in the 68K version but not in the PowerPC version. That includes the ‘CODE,’ ‘DATA,’ ‘DREL,’ and ‘ZERO’ resources. When you’re done, your resource files should look like the ones shown in Figure 8.31.



You now have a fat binary application. The PowerPC version contains all its original resources, as well as the resources that were unique to the 68K version of the program. Any resources that appeared in both versions, such as the 'DITL' and 'DLOG', were identical to one another and didn't need to be copied. Save the resource files and exit the resource editor.

The fat application will still have the name `UPPdemo1(PPC)`. Rename it so that it is obvious that it is now a fat binary. In Figure 8.32, I'm renaming the application to `UPPdemo1(fat app)`.

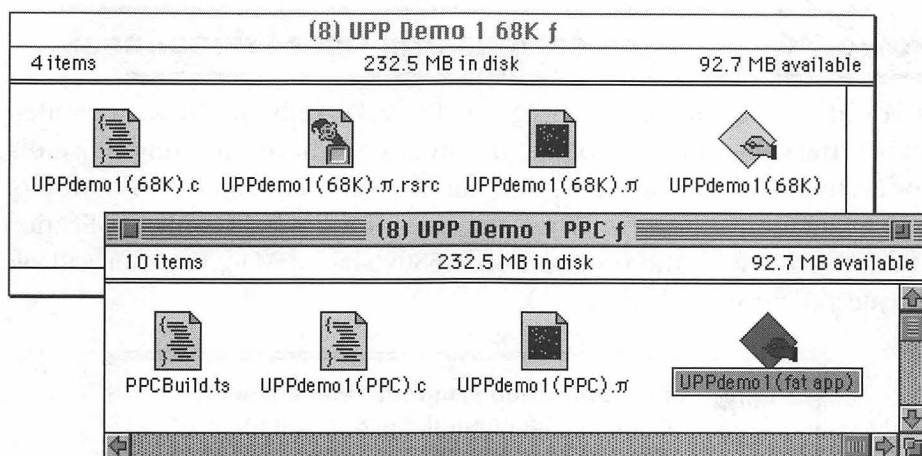


FIGURE 8.32 GIVE THE NEW FAT BINARY AN APPROPRIATE NAME.

You can test the fat application by trying to run it on both a 680x0-based Mac and a PowerPC-based Mac. Unlike the PowerPC version—which won't run on a 680x0-based Mac—this application will run on both systems.

GRACEFULLY EXITING A POWERPC-ONLY APP

Building an application using a PowerPC compiler such as the Metrowerks MW C/C++ PPC compiler or the Symantec CDK results in an application that only runs on a PowerPC-based Mac. If the program is to run on both 680x0-based Macs and PowerPC-based Macs, you'll want to turn it into a fat binary. If you know that all of the users of a program you created use Power Macs, however, you'll want to leave your application PowerPC-only. Depending on the size of the application, that strategy can save a considerable amount of disk space.

PowerPC-only Applications and User-Friendliness

If you do decide to keep a program PowerPC-only, you'll want to alert 680x0 users who attempt to use the program. There's nothing more disconcerting than having a program fail when it is launched—with little or no explanation as to why. When the user of a PowerPC-only application attempts to launch the program on a 680x0-based Mac, the user will see the alert pictured in Figure 8.33.

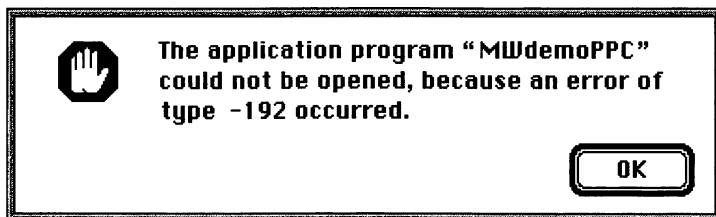


FIGURE 8.33 A POWERPC APPLICATION LAUNCHED ON A 680x0 MAC GIVES THE USER A NONDESCRIPTIVE ERROR MESSAGE.

An error of ID -192 is a “resource not found” error. You and I know that this means that the 680x0 Mac looked for a CODE resource when it attempted to launch the application. Since it's a PowerPC application, the code is in the data fork, not in 'CODE' resources. It's very likely that the user of the program won't have any idea what error -192 means. You can help the 680x0 user out by having your PowerPC-only application display an alert like the one shown in Figure 8.34. It informs the user that the program can't be run on a 680x0-based Mac. Then the program can exit. That's called a *graceful* exit, and it's a feature that's quite easy to add to your PowerPC-only applications.

Creating a fat application involves building two separate programs and then combining them into one. This is the same technique you'll use to create a PowerPC-only that displays a user-friendly alert when launched on a 680x0-based Mac. As an example, I'll start with the existing PowerPC-only version of UPPdemo1. I'll create a small 680x0 program that displays an alert, then merge the two applications.



FIGURE 8.34 THE RESULTS OF REPLACING THE SYSTEM ERROR MESSAGE WITH A MORE INFORMATIVE MESSAGE.

The 680x0 Resource File

The 68K program that is to merge with the PowerPC application has but a single purpose—to display an alert. In Figure 8.35 I’ve created a resource file that has an ‘ALRT’ resource and a ‘DITL’ resource in it. I’ve given both an ID of 129.

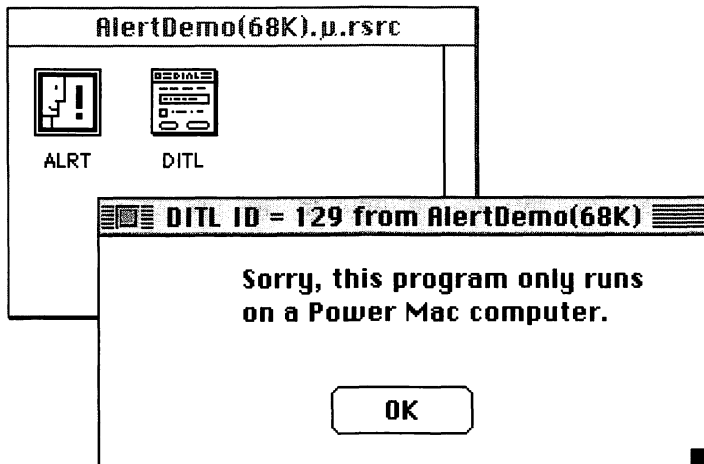


FIGURE 8.35 THE RESOURCE FILE FOR THE 68K PROGRAM THAT WILL BE ADDED TO THE POWERPC PROGRAM.

Programming the PowerPC

When ResEdit creates a new resource it usually gives the resource an ID of 128. I changed the IDs of the two resources to 129 because the resource file for the UPPdemo1 program contains a ‘DITL’ with an ID of 128—as shown in Figure 8.36. After both programs are built you’ll copy the resources from the 680x0 application into the PowerPC application. You’ll want to plan ahead so that you’ll avoid a resource ID conflict.

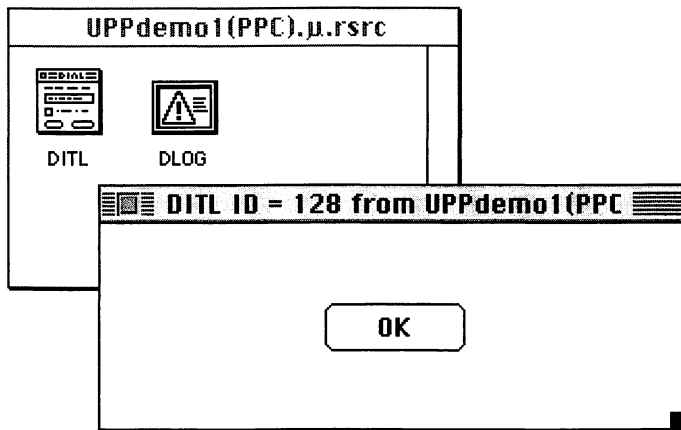


FIGURE 8.36 THE RESOURCE FILE FOR THE POWERPC PROGRAM.



NOTE

CodeWarrior uses resource files with the “μ” character in the file name—as shown in the previous two figures. If you’re using Symantec’s CDK, use the “π” character that is part of that environment’s naming convention.

The 680x0 Source Code

The source code file for the 680x0 application need do nothing but post the alert that’s defined in the resource file. Using CodeWarrior, I created a project named AlertDemo.μ and a source code file AlertDemo.c. If you’re using Symantec’s CDK, name the project AlertDemo.π. Here, in its entirety, is the source code for the 68K application.

Chapter 8 Fat Binary Applications

```
//+++++ function prototypes ++++++
void    Initialize_Toolbox( void );

//+++++ define directives ++++++
#define  ALERT_ID      129

//+++++ main ++++++

void main( void )
{
    Initialize_Toolbox();

    StopAlert( ALERT_ID, nil );
}

//+++++ initialize the Toolbox ++++++

void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}
```

Use your compiler to build the 68K application. Then quit to return to the desktop.

Copying the Resources to the PowerPC-only App

You add the 68K program to the PowerPC program in the same way you created a fat binary—you copy all the resources that are present in the 68K program but are not found in the PowerPC application. Use your resource editor to open both applications—as I’ve done in Figure 8.37.

Programming the PowerPC

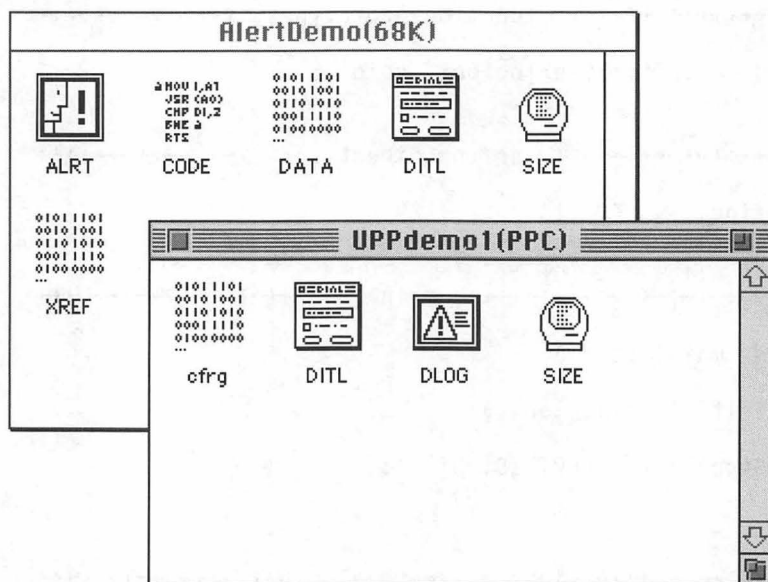


FIGURE 8.37 THE RESOURCE FILES FOR BOTH THE 68K PROGRAM AND THE POWERPC PROGRAM.

Copy the 'ALRT,' 'CODE,' 'DATA,' 'DITL,' and 'XREF' resource from the AlertDemo(68K) program and paste them into the UPPdemo1(PPC) program. Figure 8.38 shows the results of this editing. Don't forget about the 'DITL' resource. When creating a fat binary, you won't need to copy resources such as 'DITL,' 'DLOG,' and 'MENU'. That's because you're working with two versions of the same program. Here, you're combining two very different programs. The UPPdemo1(PPC) application has a 'DITL' with an ID of 128, while the AlertDemo(68K) program has a 'DITL' with an ID of 129.

Save the files and quit the resource editor. You can test the PowerPC-only application by running it on both a 680x0 machine and a Power Mac. When you run it on the 680x0-based Mac you'll see the alert pictured in Figure 8.39. When you click on the **OK** button, the application will exit and return to the desktop.

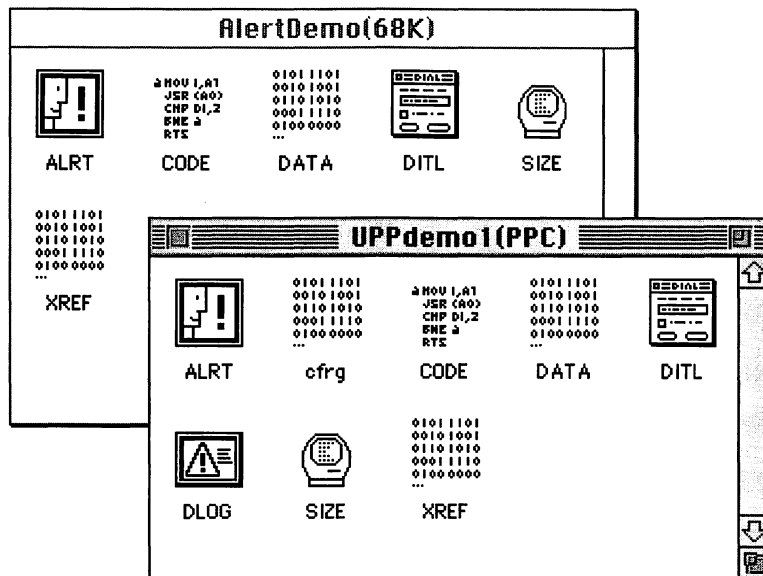


FIGURE 8.38 THE RESOURCE FILES AFTER THE 68K RESOURCES HAVE BEEN ADDED TO THE POWERPC RESOURCE FILE.



FIGURE 8.39 THE MESSAGE THE USER SEES WHEN ATTEMPTING TO RUN THE POWERPC PROGRAM ON A 68K MAC.

STRIPPING FAT APPLICATIONS

A fat application contains the code for two separate versions of a program. A fat app can therefore occupy a lot of disk space—thus the word “fat” in the name. This almost doubling of an application’s size may be worth the increased disk space—then again, it may not be. For some programs, turning the application into a fat binary may be crucial to its success—you may need to maintain the backwards compatibility that a fat app provides. There are situations, however, when you’ll know that a certain program will *only* be run on Power Macs, or only run on 680x0-based Macs. In a case such as this, there is no need to have an application that is almost double the size it needs to be. If you’ve created the program yourself, the solution is simple—compile and build it using the appropriate compiler, and don’t turn it into a fat binary. If you’ve obtained the program from an outside source, however, you’ll have to convert it yourself.



NOTE

As always, before altering a standalone application make sure to save a copy of the original program onto a floppy disk.

Converting a Fat Binary to a PowerPC Application

A fat application that will only be run on Power Macs can be converted to a PowerPC-only application to save disk space. This involves editing the applications resource fork to remove the ‘CODE’ resources. Power Mac programs don’t make use of ‘CODE’ resources, so you know you’ll be safe in doing this.

To test this conversion process, I’ve made a copy of the fat application version of UPPdemo1 that I created earlier. I’ll work with the copy and retain the original, just in case something goes wrong. You’ll find a folder named Strip to PPC *f* on the disk that came with this book. It holds a copy of both the unaltered UPPdemo1(fat app) program and a version that I’ve stripped down to a PowerPC application.

Chapter 8 Fat Binary Applications

To begin, start your resource editor. Then open the UPPdemo1(fat app) program. You'll see a resource file like the one pictured in Figure 8.40. Click the mouse on the 'CODE' resource icon and then select **Cut** from the Edit menu. The 'CODE' resources hold the bulk of the 68K version of the program. There may be other resources associated with the 68K version that aren't used by the PowerPC version—but unless you know for sure which resources these are, don't remove anything else. Since I created the original UPPdemo1(fat app), I know that the 'DATA' resources and the 'XREF' resources were copied from the 68K version of UPPdemo1. Since they only occupy a total of about 100 bytes, however, I'll leave them in. I don't want to get in the habit of deleting resources other than 'CODE' resources.

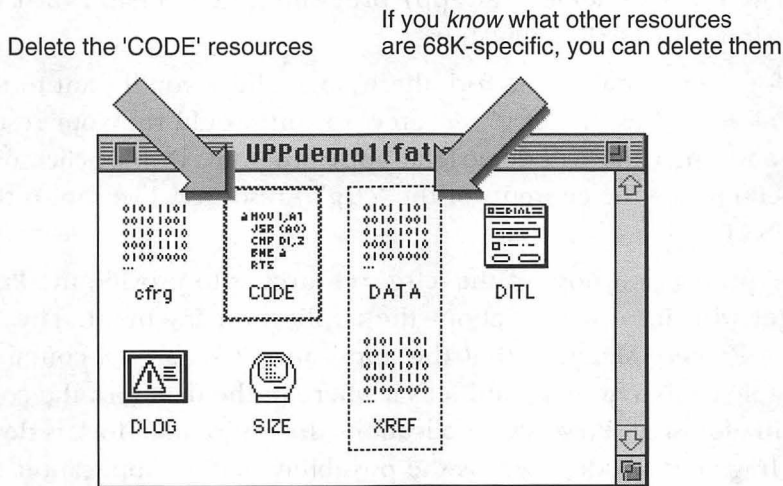


FIGURE 8.40 THE RESOURCES THAT CAN BE DELETED FROM A FAT BINARY TO MAKE IT POWERPC-ONLY.

Save the resource file and quit the editor. You now have a PowerPC-only version of the program. Give the application a more appropriate name, such as UPPdemo1(now PPC). If you now run the program on a Power Mac, it will run in fast native PowerPC mode—just as it did before. If, however, you attempt to run it on a 680x0-based Mac, you'll get the system alert that displays the -192 error.

Converting a Fat Binary to a 680x0 Application

If you've obtained a fat application that you know will only be running on 680x0-based Macs, you can remove, or "zap," the data fork. The data fork holds the PowerPC version of the program, so deleting it will greatly reduce the size of the application.

In this section I'll use the fat application version of UPPdemo1 that I created earlier in this chapter. Before I begin I'll of course make a copy of the UPPdemo1 (fat app) program and work on that copy. If something goes wrong, I'll still have the original, unaltered program. You'll find a folder named Strip to 68K *f* on the included disk. It contains a copy of the unaltered UPPdemo1 (fat app) program and a version of it that's been stripped to a 68K application.

Before deleting the data fork there's one check you'll want to make. This check involves the 'cfrg' resource, so you should run your resource editor and open the UPPdemo1 (fat app) program. Double-click on the 'cfrg' icon to see the contents of the 'cfrg' 0 resource. I've shown this in Figure 8.41.

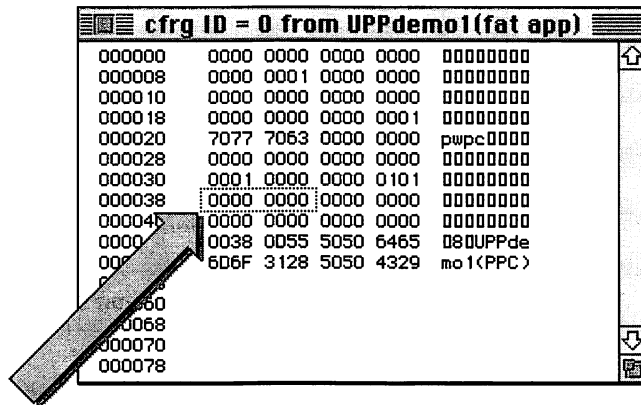
The primary purpose of the 'cfrg' resource is to provide the Process Manager with information about the application fragment. The 'cfrg' tells the Process Manager that the application's data fork contains an executable code fragment, and tells it where in the data fork the code is. While in almost all PowerPC applications the entire data fork is devoted to the fragment's code, there is the possibility that an application could use a part of the data fork for other purposes. If this is the case, the code fragment won't begin at the start of the data fork—it will be offset a number of bytes.

Before deleting the data fork you'll want to verify that the code fragment is in fact the first thing in the data fork. There are four bytes in the 'cfrg' resource that give this information. The 'cfrg' has a strictly defined format, so these bytes always appear in the same place within the resource. In Figure 8.41 I've outlined the four bytes.

The four bytes starting at 000038 in the 'cfrg' resource indicate the data fork offset at which the code fragment begins. If the four bytes pic-

Chapter 8 Fat Binary Applications

tured in Figure 8.41 all consist of zeros, the application code fragment is the first and only information in the data fork. An offset of zero means it's all right to zap the data fork. Any other numbers indicate that some information aside from the code fragment appears in the data fork. Since you don't know what this information is—and because the application may rely on the information in order to run properly—you'll want to leave the application as a fat binary.



These bytes tell where the code fragment starts in the data fork

FIGURE 8.41 THE BYTES OF A 'CFRG' RESOURCE THAT INDICATE THE RESOURCE FORK OFFSET.



NOTE

A complete description of the information in each byte of the 'cfrg' resource can be found in *Inside Macintosh: PowerPC System Software*.

After confirming that the code fragment data fork offset is 0, select **Get Info** from the File menu. If you're using ResEdit, the File menu looks like the one pictured in Figure 8.42.

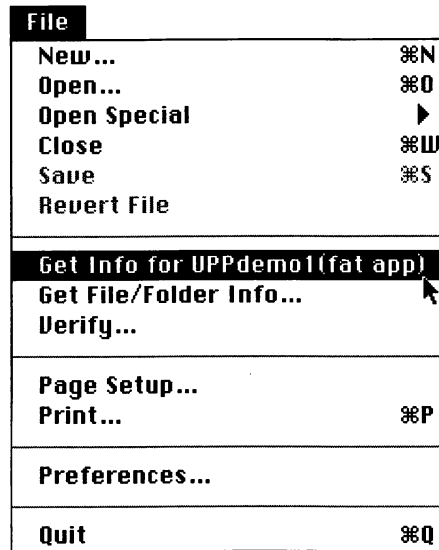


FIGURE 8.42 THE GET INFO MENU ITEM IN RESEDIT'S FILE MENU.

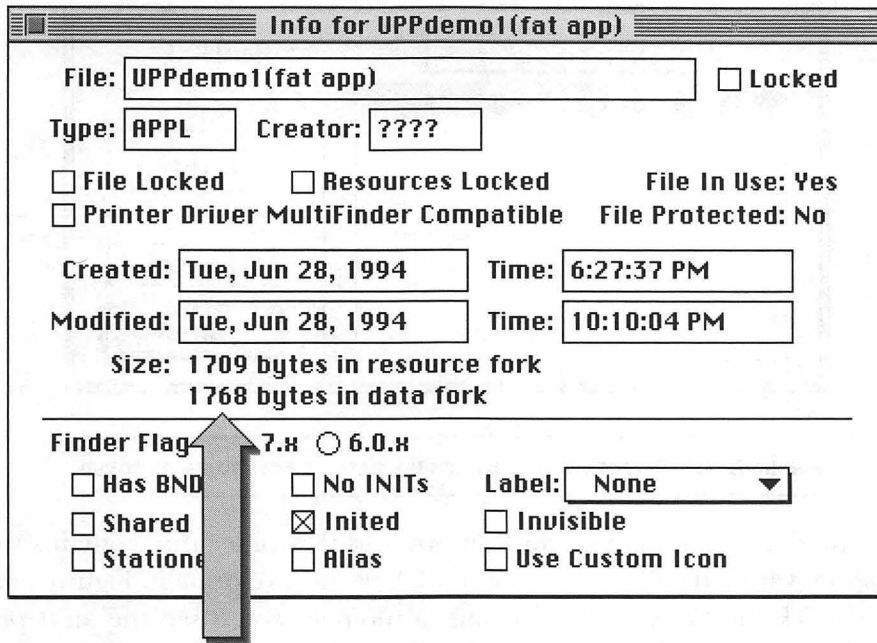
In ResEdit, selecting **Get Info** will bring up a dialog box like the one shown in Figure 8.43. Here you can see the size of the two forks that make up the application—the resource fork and the data fork.

After deleting the data fork, I'll return to ResEdit and the Get Info dialog box to take a look at the size of each fork. If all goes well, the data fork size should be 0.

To zap the data fork you'll want to use a utility designed for just that purpose. The disk that was included with this book contains just such a program. It's called DFerase, and appears in the Utility *f* folder. Quit your resource editor and go to that folder. Figure 8.44 shows the icon for the DFerase utility.

Double-click the DFerase icon to launch the program. You'll see an introductory dialog box with the program's name in it. Click the **Start** button to dismiss it. At this point the screen will be empty. Select **Open** from the File menu. You'll see a dialog box like the one shown in Figure 8.45.

Chapter 8 Fat Binary Applications



The PowerPC version of this fat binary occupies 1768 bytes

FIGURE 8.43 CHECKING THE SIZE OF THE FAT APPLICATION'S RESOURCE FORK AND DATA FORK.

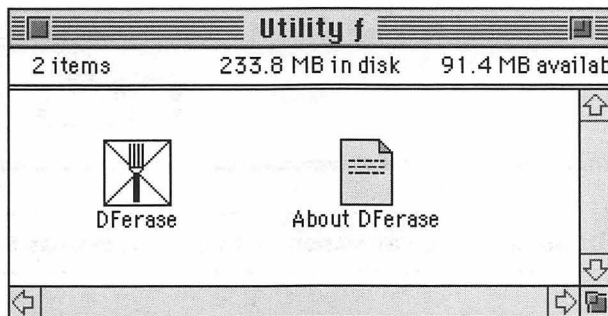


FIGURE 8.44 THE ICON FOR THE DFERASE DATA FORK ZAPPER.

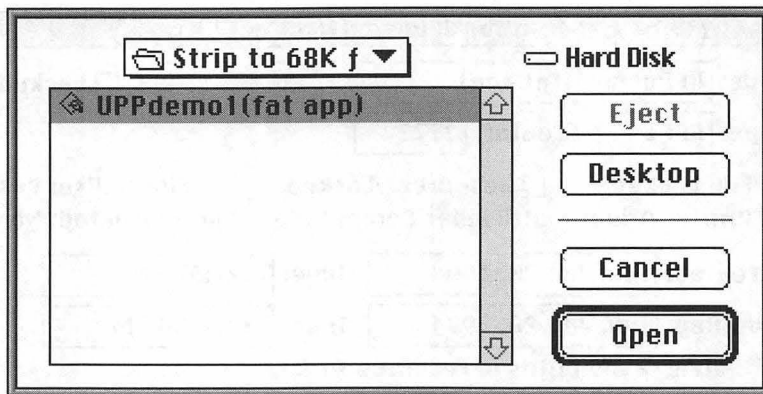


FIGURE 8.45 SELECTING THE FILE WHOSE DATA FORK IS TO BE REMOVED.

Use the dialog's pop-up menu to move into the folder that contains the program whose data fork you wish to delete—as I've done in Figure 8.45. Then click the **Open** button. In just a moment you'll see the alert pictured in Figure 8.46. Click the **OK** button to dismiss the alert, then select **Quit** from the File menu to exit DFERase.

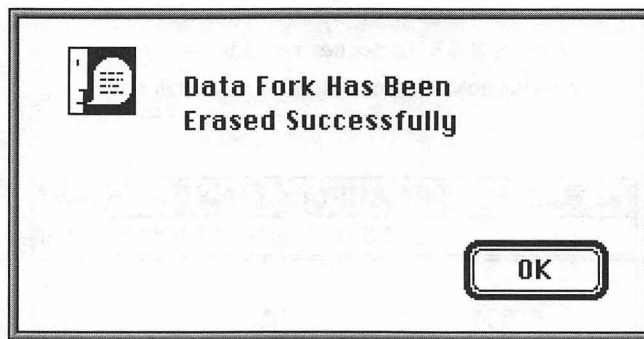
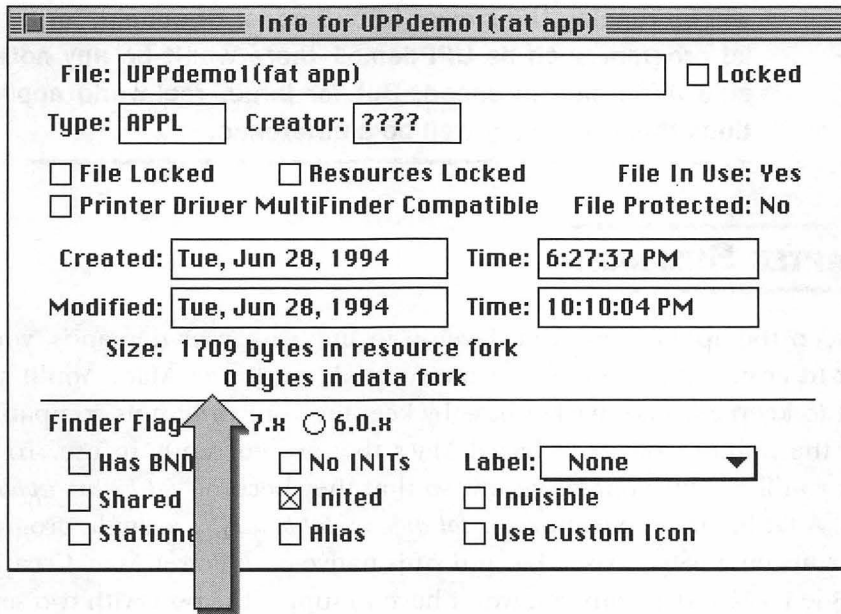


FIGURE 8.46 DFERASE GIVES CONFIRMATION THAT THE DATA FORK HAS BEEN DELETED.

The last step in turning the fat binary into a 68K application is to remove the 'cfrg' resource. Again run your resource editor. Open the UPPdemo1(fat app) program. Before deleting the 'cfrg', select **Get Info**

from the File menu. You'll see that the resource fork hasn't changed in size, but the data fork size is now 0—just as you want. This is shown in Figure 8.47. After confirming this, close the Get Info window.



After zapping the data fork,
the data fork size is zero

**FIGURE 8.47 USING RESEDIT'S GET INFO DIALOG
TO VERIFY THAT THE DATA FORK HAS BEEN REMOVED.**

Now you'll delete the 'cfrg' resource. When running any application on a 680x0-based Mac, the 'cfrg' resource—if present—is ignored. But if you attempt to run this modified application on a Power Mac, the system will examine the 'cfrg' resource and assume it is working with a PowerPC application. Since you've deleted the data fork, this is a situation you want to avoid. Click once on the 'cfrg' resource icon, then select **Cut** from the Edit menu.

Select **Save** from the File menu and quit the resource editor. You now have a 68K application from what was a fat binary. Give the program an appropriate name, such as UPPdemo1(now 68K).

Programming the PowerPC



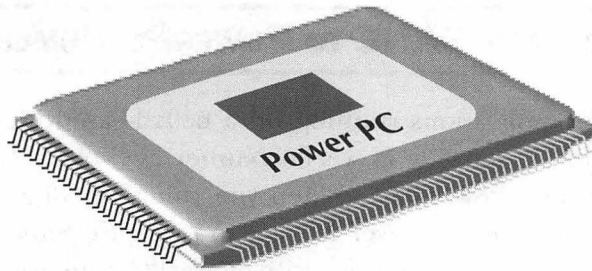
You can still run the new 68K version on a Power Mac—just as you can run any 68K application on a Power Mac. It just won't be running native PowerPC code anymore. Instead, it will be running the slower 680x0 code. Of course, for a trivial program such as UPPdemo1 there won't be any noticeable difference in speed. But for large, real-world applications there may very well be a difference.

CHAPTER SUMMARY

To keep the applications you develop in line with user demands, you'll want to ensure that they run in native mode on Power Macs. You'll also want to keep a broad market base by keeping your programs compatible with the millions of 680x0-based Macs that are currently in use. To do that, you'll modify your programs so that they become *fat binary applications*. A fat binary application, or *fat app*, or *fat binary*, is a single program that runs on a 680x0-base Mac and runs native on a Power Mac. Creating a single fat binary means you won't have to supply the user with two separate versions of your program.

For those occasions when you know a program will be run only on a Power Macintosh, you'll want to make sure that users of 680x0-base Macs understand why their attempts to run your program fail. Using the principles involved in creating a fat application, you can create a program that runs as planned on a Power Mac, but displays an informative alert when an attempt is made to run it on a 680x0-based Mac.

A fat binary is large in size—it takes up much more disk space than its PowerPC-only version. If own a fat application developed by someone else, you can reduce the amount of disk space it occupies. Simply convert it to a PowerPC-only application by stripping the application of its data fork.



CHAPTER 9

THE POWERPC NUMERICS ENVIRONMENT

Whenever you write code that performs a mathematical operation, you're using a numeric environment. Typically, this environment consists of a set of routines that are a part of the programming language you use. If your 680x0 code contains floating-point operations, your application is making use of routines found in the Standard Apple Numerics Environment—better known as SANE.

The PowerPC-based Macintoshes don't use SANE. Instead, Power Macs use PowerPC Numerics. PowerPC Numerics is an environment used to allow quick and accurate computation of floating-point expressions on Power Macintosh computers. If your PowerPC code performs floating-point operations, then you're using the PowerPC Numerics environment.

In this chapter you'll see how PowerPC Numerics differs from SANE. Because those differences will need to be addressed when you port 680x0 code to native PowerPC code, numerics porting considerations are also covered here.

SWITCHING FROM SANE TO POWERPC NUMERICS

Macintosh applications running on a 680x0-based Macintosh make use of SANE—never PowerPC Numerics. PowerPC Numerics is only available on a Power Mac. When you run a 680x0 application that hasn't been ported on a Power Mac, it too will use SANE. Since older 680x0 applications aren't familiar with PowerPC Numerics, SANE compatibility is a necessity. So when is PowerPC Numerics used? When you run a native PowerPC application on a Power Macintosh. Figure 9.1 shows these three scenarios.

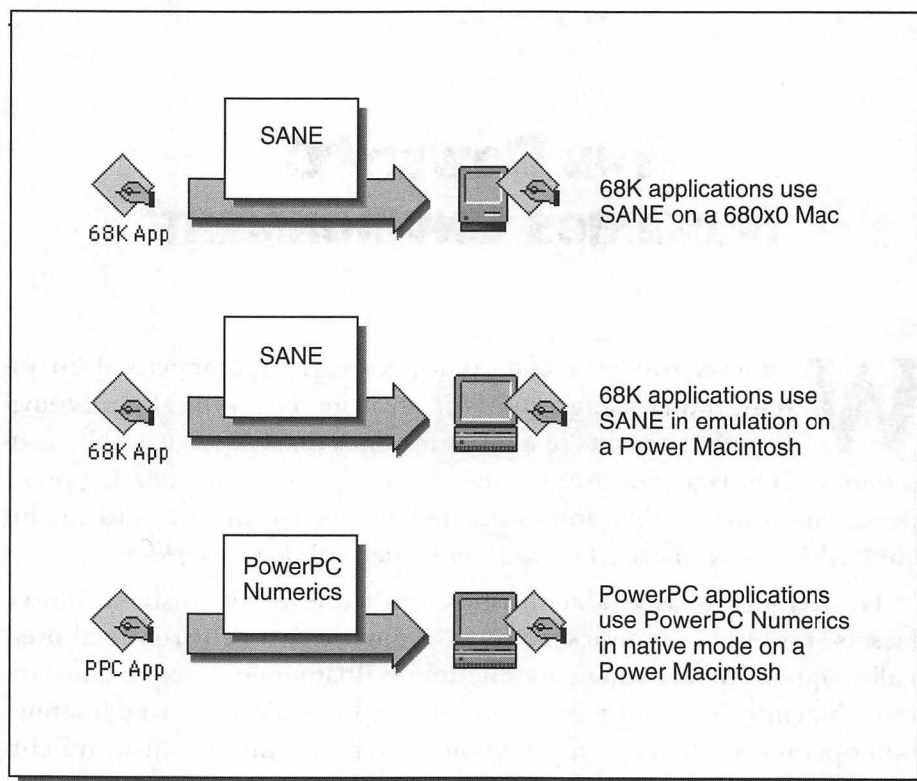


FIGURE 9.1 SCENARIOS FOR USING SANE AND POWERPC NUMERICS.

Chapter 9 The PowerPC Numerics Environment

From Figure 9.1 you can see that porting code that uses SANE to PowerPC Numerics code is optional—680x0 applications that use SANE will run on a PowerPC. So why expend effort on porting properly functioning SANE code to PowerPC Numerics code? Once again, the word “native” appears in the answer. These 680x0 applications will use SANE on a PowerPC, but they will use it in emulation mode. Porting this code will allow it to run in the much faster, native PowerPC Numerics environment. So for PowerPC development, you won’t use SANE. As Metrowerks very bluntly states in its documentation, “SANE is dead.”

Why the need for an entirely new numerics system? Because the PowerPC microprocessor differs so greatly from the Motorola 680x0 processors. SANE is based on an extended 80-bit data format. The PowerPC microprocessor relies on a 64-bit data format called double. While possible, attempting to force the double-based PowerPC to manipulate data that is in an 80-bit format would be very inefficient.

Coming from the 80-bit data format that’s available on the 680x0 processors to the 64-bit data format on the PowerPC seems like a step down. In most instances however, 64-bits is sufficient. And when it isn’t, Power Numerics supports a 128-bit data format called double-double.

POWERPC NUMERICS DATA FORMATS

PowerPC Numerics makes use of three floating-point data formats: the single format, the double format, and the double-double format. Just as the 680x0 floating-point format is represented by the Pascal *real* data type and the C *float* data type, each of the PowerPC Numerics formats has a data type representation. The single format is represented by the *float* data type, the double format is represented by the *double* data type, and the double-double format is represented by the *long double* data type. Here are the declarations of a variable of each of these types:

```
float      the_single;
double     the_double;
long double the_double_double;
```

The Single Format

The single format is represented by the *float* data type. A PowerPC Numerics *float* occupies four bytes. These 32 bits provide a range of $-3.4\text{E}+38$ to $+3.4\text{E}+38$ with seven to eight digits of precision.

The Double Format

The PowerPC Numerics double format is represented by the *double* data type. The *double* occupies eight bytes. The 64 bits of the double format give a range of $-1.8\text{E}+308$ to $+1.8\text{E}+308$ with fifteen to sixteen digits of precision.

The Double-Double Format

The double-double format is represented by the *long double* data type. A PowerPC Numerics *long double* occupies 16 bytes. These 128 bits provide the same range as the double format: from a minimum negative value of $-1.8\text{E}+308$ to a maximum positive value of $+1.8\text{E}+308$. The difference between a double format and a double-double format is not in the size of the number that the types hold, but rather in the precision that the two types provide. While the double format yields numbers with fifteen to sixteen digits of precision, the double-double format will provide no less than thirty-two decimal digits of accuracy.



NOTE

Since the double-double format offers no greater range of values, in most instances you'll be using the double format. Computations are performed much more quickly using the eight byte double than the sixteen byte double-double.

Numeric Data Format Summary

Table 9.1 is a summary of the attributes of the three PowerPC Numerics floating-point data types. Of the three types, you'll find that the double

Chapter 9 The PowerPC Numerics Environment

data type provides the best compromise in terms of precision, range, and speed.

TABLE 9.1 POWERPC NUMERICS DATA FORMAT SUMMARY.

	float	double	long double
Size (in bits)	32	64	128
Precision (in digits)	7 — 8	15 — 16	>= 32
Maximum positive value	+3.4E+38	+1.8E+308	+1.8E+308
Minimum negative value	—3.4E+38	—1.8E+308	—1.8E+308

NUMERICS LIBRARIES AND THE POWERPC

If you've ever used SANE for 680x0 development, you've included the SANE.h header file in your source code:

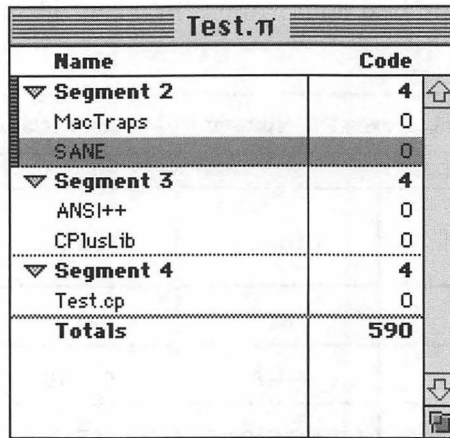
```
#include <SANE.h>
```

In your PowerPC source code, you'll want to substitute the fp.h header file for the SANE.h header file:

```
#include <fp.h>
```

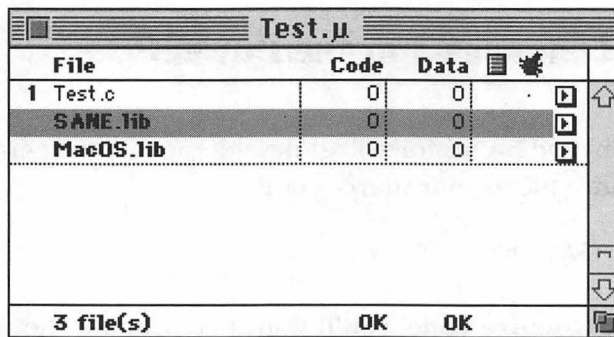
If you used SANE functions, then besides the inclusion of the SANE.h header file, you also added the SANE library to your 680x0 project. Figure 9.2 shows a Symantec project window that includes the SANE library. Figure 9.3 shows a CodeWarrior 68K project window with the Metrowerks version of the SANE library, SANE.lib, included.

Programming the PowerPC



Name	Code
▼ Segment 2	4
MacTraps	0
SANE	0
▼ Segment 3	4
ANSIC++	0
CPlusLib	0
▼ Segment 4	4
Test.cp	0
Totals	590

FIGURE 9.2 INCLUDING THE SANE LIBRARY IN A SYMANTEC PROJECT.



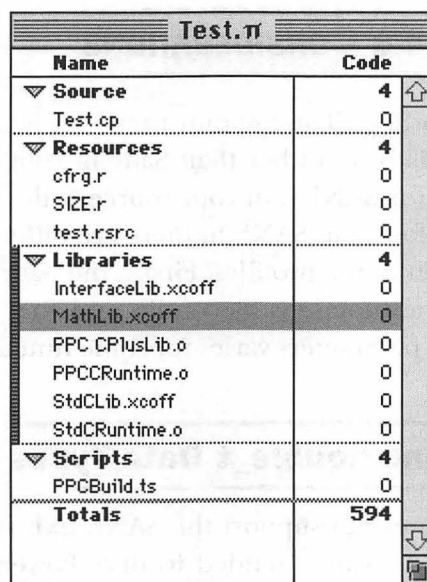
File	Code	Data
1 Test.c	0	0
SANE.lib	0	0
MacOS.lib	0	0

3 file(s) OK OK

FIGURE 9.3 INCLUDING THE SANE LIBRARY IN A METROWERKS PROJECT.

Now, for PowerPC development, you'll use the MathLib library rather than a SANE library. If you use the Symantec CDK, and you use the method of copying an existing project folder to use as the basis of a new project, then your project will already have the MathLib in it. Figure 9.4 shows a typical Symantec PowerPC project with the MathLib.

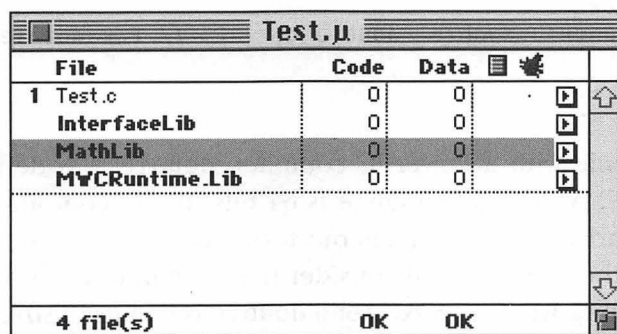
Chapter 9 The PowerPC Numerics Environment



Name	Code
▼ Source	4
Test.cp	0
▼ Resources	4
cfrg.r	0
SIZE.r	0
test.rsrc	0
▼ Libraries	4
InterfaceLib.xcoff	0
MathLib.xcoff	0
PPC CPPlusLib.o	0
PPC CRuntime.o	0
StdCLib.xcoff	0
StdCRuntime.o	0
▼ Scripts	4
PPCBuild.ts	0
Totals	594

FIGURE 9.4 INCLUDING THE MATHLIB LIBRARY IN A SYMANTEC PROJECT.

Chapter 6 made the recommendation that CodeWarrior users always add the same three libraries to their PowerPC projects. One of those libraries was MathLib. Figure 9.5 shows the project window of an MW C/C++ PPC project with the MathLib included.



File	Code	Data
1 Test.c	0	0
InterfaceLib	0	0
MathLib	0	0
MWCRuntime.Lib	0	0
4 file(s)		

FIGURE 9.5 INCLUDING THE SANE LIBRARY IN A METROWERKS PROJECT.

NUMERICS PORTING CONSIDERATIONS

Unfortunately, porting floating-point numerics is not quite as easy as just including MathLib rather than Sane in your PowerPC projects, and substituting fp.h for SANE.h in your source code. While fp.h has just about every function found in SANE.h, there are differences between the function headers listed in the two files. Firstly, the data type of some parameters varies for some functions. Secondly, and to a lesser degree, the number and order of parameters varies for some functions.

The extended and double_t Data Types

PowerPC Numerics does not support the SANE extended floating-point format. Instead of the 80-bit extended format, PowerPC Numerics supports the 64-bit double_t format. You'll find the double_t format defined in the Types.h universal headers file such that it will be interpreted as 64 bits by both PowerPC compilers and 680x0 compilers. Here's a part of that definition:

```
#ifdef applec
    typedef long double double_t; // 68K long double is 64
                                bits

#elif powerc
    typedef double double_t;      // PPC double is 64 bits

#endif
```

If you compile with a PowerPC compiler, double_t is defined to be of type double. A PowerPC double is 64 bits. If you compile with a 680x0 compiler and the compiler encounters a variable that is declared as a double_t, the compiler will consider it a long double. A double on the 680x0 is 32 bits, while a 680x0 long double is 64 bits in size.

Since PowerPC Numerics doesn't use the extended format, and SANE relies heavily upon it, the conversion from extended to double_t may be your biggest numerics porting concern. You'll want to search

Chapter 9 The PowerPC Numerics Environment

your source code for the word “extended” and replace all occurrences of it with “double_t.” As you do so, though, you’ll also want to have both the SANE.h and fp.h header files open. When you encounter a SANE function that has a parameter or return type of extended, you’ll want to compare the SANE.h version of the function header with the fp.h version. For some functions, the number or order of the arguments may differ.

As an example, consider the SANE function `scalb()`. This function multiplies x times 2^n , where x and n are values passed in as parameters. In SANE.h, `scalb()` is defined as:

```
extended scalb( short, extended );
```

Here’s a 680x0 code snippet that makes a call to `scalb()`:

```
short    n = 2;      // first parameter to scalb()
extended x = 3;      // second parameter to scalb()
extended result;     // hold result returned by scalb()

result = scalb( n, x ); //  $x * 2^n = 3 * 2^2 = 3 * 4 = 12$ 
```

PowerPC Numerics also supports the `scalb()` function, but the arguments and return type are different than those found in the SANE implementation of `scalb()`. Additionally, the ordering of the two arguments is reversed. In fp.h, `scalb()` is defined as:

```
double_t scalb ( double_t x, long int n );
```

In PowerPC Numerics, x is listed first and n is listed second. That’s the opposite of the SANE definition of the same function. Here’s a PowerPC snippet of code that makes a call to `scalb()`:

```
double_t x = 3;      // first parameter to scalb()
long int n = 2;      // second parameter to scalb()
double_t result;     // hold result returned by scalb()

result = scalb( x, n ); //  $x * 2^n = 3 * 2^2 = 3 * 4 = 12$ 
```

Eliminate the comp Data Type

When a very high degree of precision is necessary, some SANE routines make use of the comp data type. The comp data type offers 64 bits of precision. Since PowerPC Numerics offers no comp type, you'll have to replace any usage of this type with a PowerPC Numerics type. The PowerPC Numerics double offers 53 bits of precision—that translates to 15 to 16 decimal digits of accuracy. You should search your source code listings for each occurrence of comp and, where acceptable, replace these references with double. If still more accuracy is needed you can use the PowerPC Numerics long double type.

Be Aware of How Expressions Are Evaluated

In SANE, all floating-point operations are performed using extended precision. PowerPC Numerics has no extended equivalent. Operations involving large values that compiled properly on a 680x0 compiler may not yield the correct results on a PowerPC compiler. In particular, you'll have to be watchful for midexpression overflow. Consider this snippet:

```
double d_1 = 1.6E+308; // max double is 1.8E+308
double d_2 = 1.4E+308; // max double is 1.8E+308
double answer;

answer = ( d_1 + d_2 ) / 2; // (3.0E+308)/2 = 1.5E+308
```

Because midexpression (d_1 + d_2) has a value larger than the maximum value a double can hold (1.8E+308), a 680x0-base Macintosh will use the SANE extended data type to hold the result of the midexpression. After the midexpression value of 3.0E+308 is divided by 2, the result will be stored in the double variable answer.

On a PowerPC, the above snippet will generate a midexpression overflow, and the final result will be unusable. PowerPC Numerics has no extended data type, so the midexpression value of 3.0E+308 cannot be properly stored. On a PowerPC, you'll want to evaluate numeric opera-

Chapter 9 The PowerPC Numerics Environment

tions and, where necessary, separate one involved operation into two or more simpler operations. For the above snippet, you could rewrite the one numeric operation into three separate ones:

```
double d_1 = 1.6E+308;
double d_2 = 1.4E+308;
double temp_1;           // divide each value separately,
double temp_2;           // holding the results in temp
                        // vars

double answer;

temp_1 = d_1 / 2;         // (1.6E+308)/2 = 0.8E+308
temp_2 = d_2 / 2;         // (1.4E+308)/2 = 0.7E+308
answer = temp_1 + temp_2; // (0.8E+308 + 0.7E+308) =
                        // 1.5E+308
```

PowerPC compilers define a *minimum evaluation format* that is used in all expression evaluations performed by that compiler. The minimum evaluation format specifies the least precision that will be used in all expression evaluations. The compiler designers will implement this minimum evaluation format as any one of the three PowerPC Numerics data format types: single, double, or double-double.

If a compiler has a minimum evaluation format of double, then all operands of less precision than a `double` will be promoted to a `double` during evaluation of that expression. Consider the expression in this snippet:

```
float s_1 = 1.5E+20;
float s_2 = 1.0E+20;
double answer;

answer = s_1 * s_2;
```

If the minimum evaluation format is double, then each of the two single format variables (C `float` types) will be temporarily promoted to `double` during expression evaluation, and the result of their multiplication together will also be temporarily held in a `double`. This is shown in Figure 9.6.

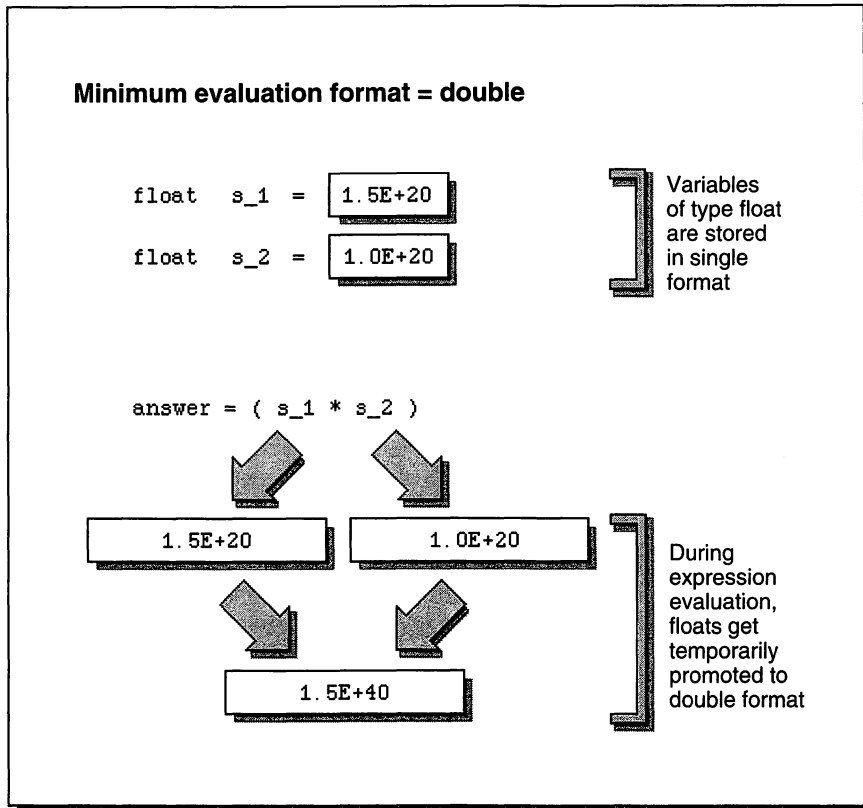


FIGURE 9.6 WHEN THE MINIMUM EVALUATION FORMAT IS DOUBLE, SINGLE VALUES ARE PROMOTED TO DOUBLE.

If the minimum evaluation format is single rather than double, then each of the two single format variables in the previous snippet will remain in single format during expression evaluation. Because intermediate operations aren't temporarily held in a larger data format, the potential for overflow increases. In Figure 9.7, the same values that were used in the example for a minimum evaluation format of double are used. Yet here, the result of the multiplication operation will be garbage. That's because while each of the two float variables is in range, the result of their multiplication together (1.5E+40) exceeds the maximum value that the single format can hold (1.8E+38).

Chapter 9 The PowerPC Numerics Environment

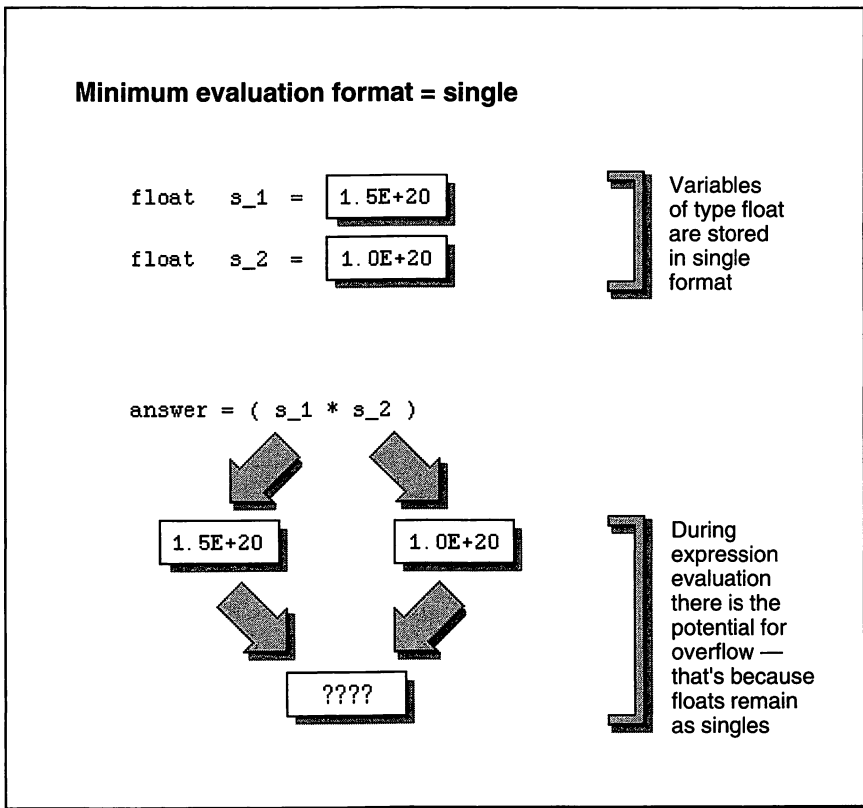


FIGURE 9.7 WHEN THE MINIMUM EVALUATION FORMAT IS SINGLE, THE POTENTIAL FOR OVERFLOW INCREASES.

From the preceding discussion and figures, it seems that the higher, or “wider”, the minimum evaluation format, the better. After all, overflow resulting from expression evaluation is obviously reduced when the format is double as opposed to a format of single. There are advantages, however, to using the single format.



NOTE

Though the double-double format exists for high-precision numerics, the PowerPC Numerics are based on the single and double formats. Thus Power Macintosh compilers

Programming the PowerPC

implement either the single or double format for the minimum evaluation format.

If an expression contains variables of both single and a double format, then the evaluation of that expression will take place in a double format. This is true regardless of the minimum evaluation format. Consider the expression listed here:

```
float   s_1  = 1.5E+20;
double  d_1  = 1.0E+50;
double  answer;

answer = s_1 * d_1;
```

Here, the intermediate result of (`s_1 * d_1`) is held as a *double*. That's because a `double`, `d_1`, is involved in the expression. So while the result of the above multiplication (`1.5E+70`) would overflow a single, the operation will still be successful because a `double` is used to hold the intermediate result. In instances such as this, the minimum evaluation format isn't a consideration.

The single minimum evaluation format yields the fastest results in the evaluation of single-precision expressions. And, expressions that involve a mix of single and double types can be evaluated accurately due to the compiler's ability to store intermediate results in the more precise double format.

The double minimum evaluation format has the advantage of preventing midexpression evaluation overflow. However, expressions involving only single format variables will be evaluated less efficiently than they would be with a single minimum evaluation format compiler. That's because the single types will be promoted to double types before the evaluation—even if the single-precision variables are small and the potential for overflow doesn't exist.



Remember, the choice of minimum evaluation format is left to the discretion of each compiler manufacturer. As of this writing, the Metrowerks PPC compiler uses a minimum

Chapter 9 The PowerPC Numerics Environment

evaluation format of single, while the Symantec CDK uses a minimum evaluation format of double.

As you write PowerPC code, you should keep in mind the different way PowerPC Numerics handles numeric operations. You'll also want to closely examine numerical operations in your 680x0 code that is being ported to PowerPC code. Keep a watchful eye open for overflow conditions in both the final value of an expression and in intermediate results.

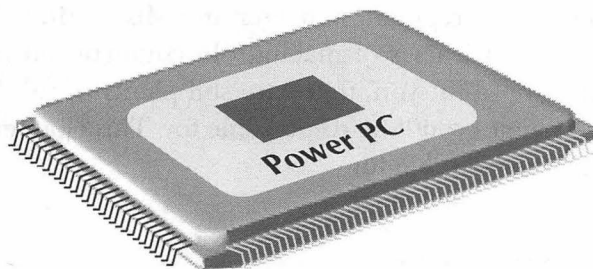
CHAPTER SUMMARY

A numerics environment defines the way a computer performs numeric operations. On a 680x0-based Macintosh, SANE is that environment. On a PowerPC-based Macintosh, the environment is PowerPC Numerics. The evaluation of any floating-point expression on the Power Macintosh involves the PowerPC Numerics environment.

PowerPC Numerics makes use of three floating-point data formats—each represented by a C data type. The single format, the double format, and the double-double format are represented by the `float`, the `double`, and the `long double`, respectively. The `float` is a 4-byte data type, while the `double` and `long double` each occupy 8 bytes. Though the range of the `double` and the `long double` is the same, the `long double` has greater precision. The trade-off is that the PowerPC cannot manipulate `long double` variables as quickly as it can `double` variables. Since the great precision of the `long double` is seldom needed, the `double` data type is the most commonly used PowerPC Numerics data type.

If you used SANE routines on a 680x0-based Macintosh, you included the `SANE.h` header file in your source code and the SANE library in your project. For the Power Mac, you'll instead include the `fp.h` header file and the MathLib library. While most of the routines in `fp.h` match those found in `SANE.h`, they are not all identical. You'll keep this in mind as you port 680x0 source code to native PowerPC code. Another porting consideration is that Power Numerics doesn't define the 80-bit extended data type found in SANE.

Compilers define a minimum evaluation format to determine how midexpression evaluation is stored in memory. A format of type double means that operands of type single will be promoted to double before a numerical operation is performed. This decreases the likelihood of overflow. A minimum evaluation format of single means operands of type single won't be promoted. Though the chances of midexpression overflow increase with this format, the speed at which numeric operations can be performed also increases.



CHAPTER 10

PORTING CODE TO NATIVE POWERPC

Almost any program written for a 680x0-based Mac will run on a PowerPC. Apple put a great effort into keeping existing code compatible by creating the 68LC040 Emulator that's built into every Power Mac. So why bother expending the effort to port 680x0 code to PowerPC code? To avoid the 68LC040 Emulator. While 680x0 applications will run on a Power Mac, they do so at the cost of constantly using the software emulator—slowing things down considerably. To take advantage of the greater speed of the PowerPC chip, you'll want to port your code and then recompile it so that your program's instructions become native to the PowerPC chip.

Much of the work involved in turning 680x0 code into PowerPC code is handled by the compiler and linker that make up your development sys-

Programming the PowerPC

tem, and by the Code Fragment Manager and Mixed Mode Manager. But while many of the intricacies of making old code run on new machines has been taken care of for you, there are still plenty of subtle—and a few not-so-subtle—changes you'll be responsible for. This chapter covers those changes you'll be responsible for.

PORTING PREPARATION

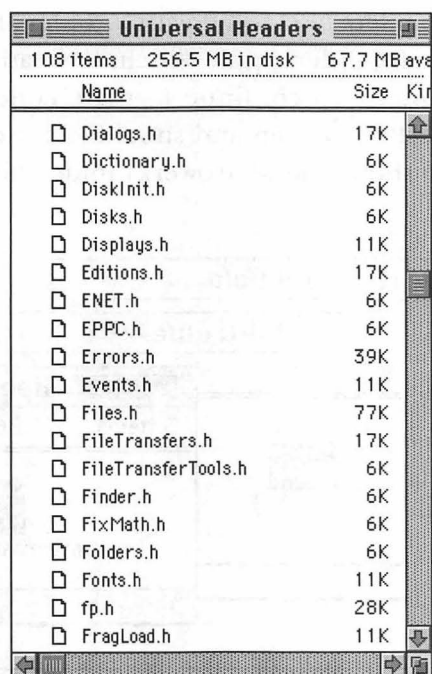
Before beginning the port of your 680x0 source code to PowerPC code, you'll want to take care of a couple of preliminary matters. The first task is easy—make sure your development environment is making use of Apple's universal header files. The second may or may not be easy—it depends on whether you have assembly language included in your 680x0 source code.

Use the Universal Header Files

First and foremost, make sure you're using the universal headers—sometimes referred to as the universal interface files. The header files are necessary if you're going to port to PowerPC. They hold information about universal procedure pointers and other PowerPC-specific data types. Figure 10.1 shows a few of the hundred-plus header files. Many of the file names, such as `Dialogs.h` and `Events.h`, should look familiar to you. Other files, such as `FragLoad.h` and `fp.h` (discussed in Chapter 9), are new to the PowerPC.

If you're using the Symantec CDK, you don't have to do anything special to make use of these header files. Symantec gives you both the older Apple `#includes` and the universal header files in separate folders. The compiler will use the headers that appear in whichever folder *doesn't* have its name nested in parentheses—as shown in Figure 10.2.

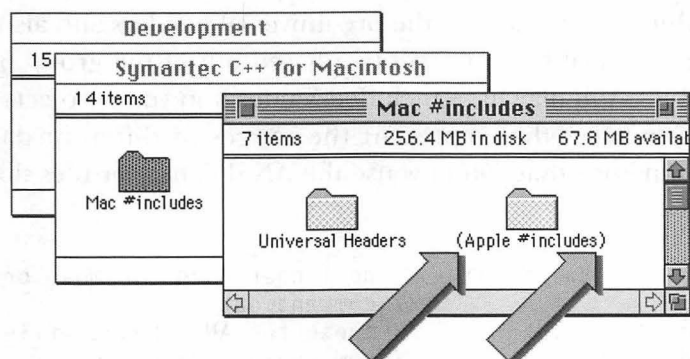
Chapter 10 Porting Code to Native PowerPC



The screenshot shows a window titled "Universal Headers" with a list of header files. The window header also displays "108 items", "256.5 MB in disk", and "67.7 MB available". The list has columns for "Name", "Size", and "Kind".

Name	Size	Kind
Dialogs.h	17K	File
Dictionary.h	6K	File
DiskInit.h	6K	File
Disks.h	6K	File
Displays.h	11K	File
Editions.h	17K	File
ENET.h	6K	File
EPPC.h	6K	File
Errors.h	39K	File
Events.h	11K	File
Files.h	77K	File
FileTransfers.h	17K	File
FileTransferTools.h	6K	File
Finder.h	6K	File
FixMath.h	6K	File
Folders.h	6K	File
Fonts.h	11K	File
fp.h	28K	File
FragLoad.h	11K	File

FIGURE 10.1 SOME OF THE HEADER FILES FROM THE UNIVERSAL HEADERS.



Nesting this folder's name between parentheses forces the Symantec compiler to use the universal header files

FIGURE 10.2 SYMANTEC SUPPLIES BOTH THE OLDER INTERFACE FILES AND THE NEW UNIVERSAL HEADERS.

Programming the PowerPC

The Metrowerks compilers also come with a folder that houses the universal header files. The older Apple `#include` header files aren't supplied. Even if you intend to continue to write code that you feel will never run on a PowerPC, you can and should still use the new universal headers. Figure 10.3 shows the Metrowerks folder hierarchy of the universal headers.

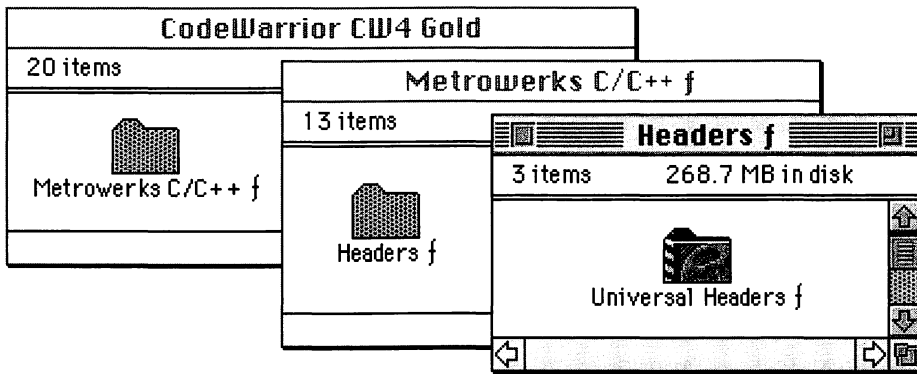


FIGURE 10.3 METROWERKS SUPPLIES THE UNIVERSAL HEADERS WITH ITS COMPILERS.

One particular header file requires a special note. The header file `Values.h`, which was present in the pre-universal headers and also appears among the universal headers, will be phased out of the group of universal header files. You may have included `Values.h` in your projects in order to access its macros that represent the ranges of different data types. Apple recommends that you now use the ANSI C header files `float.h` and `limits.h`:

```
// #include <Values.h>      no longer used - remove or
                           // comment out
#include <float.h>          // these two ANSI header files
#include <limits.h>         // now replace Values.h
```

While the `float.h` and `limits.h` files hold macros that define type ranges, the macro names differ slightly. So besides changing your `#include` directives, you'll have to be aware that you may encounter "XXXX undefined" errors. For example, if in the past you used `MAXINT`, now you'll use

INT_MAX. For a complete listing of both the old and new macro names, open and examine the universal header version of the Values.h header file. While the file doesn't contain the macros themselves, it does have a commented section that lists the old Values.h names and their new ANSI C equivalents.

Change Assembly Code to C Code

Assembly language fanatics will not be pleased with Apple's very strong recommendation that all assembly language code be rewritten as C source code. But this step is a necessary one. The PowerPC compilers do not use the `asm` directive that is found in THINK C. The `asm` directive allows assembly language to appear along with C language code in the same source code file.

Assembly routines are usually added to a C program when the programmer feels that speed is of the utmost importance. Assembly language code compiles into executable code that is faster than the executable code that results from compilation of high-level language code. While execution speed increases due to the use of assembly language may have been significant on CISC processors, that speed improvement has diminished to almost zero on the PowerPC microprocessor. Optimizing compilers—like the Metrowerks MW C/C++ PPC compiler—generate executable code that is *fast*.

Assembly language programmers would have to expend a great deal of effort hand-optimizing their assembly code for it to be able to match or beat the speed of code generated by an optimizing PowerPC compiler.



For you assembly die-hards, Apple does have a PowerPC Assembler available. It comes as part of the Apple RISC Software Developer Kit (SDK). This development environment also includes an ANSI-compliant C/C++ compiler, a two-machine debugger, and various other tools and documentation.

ANSI C AND THE POWERPC

While the PowerPC chip is new, the RISC technology on which it is based is not. Before the PowerPC microprocessor there were other RISC microprocessors, and, of course, other RISC C language compilers. The current PowerPC compilers are derived from these non-Macintosh compilers. So, like those compilers, the PowerPC compilers are based on the ANSI C standards. As such, these Macintosh PowerPC compilers do not allow for some of the features that traditional Macintosh C compilers have. Before recompiling your 680x0 source code to PowerPC code, you'll want to remove certain non-ANSI code that was accepted by 680x0 compilers, but won't be accepted by PowerPC compilers.

Change `int` Variables to Other Integral Types

A development environment has the ability to change the amount of memory that an integer will occupy. Compilers such as THINK C have a preference setting that allow the programmer to indicate whether each integer should occupy 2 bytes or 4 bytes. Figure 10.4 shows the THINK C dialog box that's used for this purpose.

While Symantec lets the programmer make the decision as to the size of an integer, other development environments—like Apple's Macintosh Programmer's Workshop (MPW) and the PowerPC compilers—fix the size of an integer at 4 bytes. This variance in the treatment of the integer data type make it a poor candidate for portability.

While source code that contains variables of type `int` will of course compile on a PowerPC compiler, you run the risk of introducing bugs in your PowerPC version of the application you're porting. Introduce a bug? How so? Consider this simplistic example:

```
int  score;
:
:
:

if ( score > MAXINT )
    Display_Number_Too_Big_Message();
```

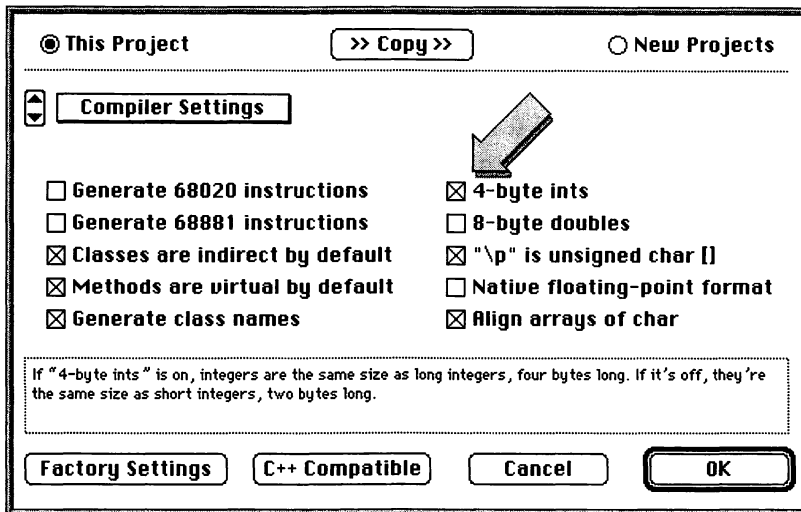


FIGURE 10.4 YOU CAN CHANGE THE SIZE OF THE `INT` DATA TYPE IN THE **THINK C** ENVIRONMENT.

If this snippet was written in a 2-byte `int` environment, then the programmer's intentions were for the error message to be displayed if `score` exceeded 32,767—the value that `MAXINT` has in a 2-byte `int` environment. The programmer, in preparation for the port to PowerPC, faithfully changes from the `Values.h` header file to the `limits.h` header file. He then looks at the table in the new version of `Values.h` to see that the `MAXINT` macro he used with his 680x0 compiler is now named `INT_MAX` in the `limits.h` header. So he changes his `if` statement to read as follows:

```
int  score;
:
:
:

if ( score > INT_MAX )
    Display_Number_Too_Big_Message();
```

The result? Now the conditional test of the `if` statement will not pass until `score` has a value of 2,147,483,647—the maximum value of the 4-byte integer that variable `score` has become in the PowerPC program.

Programming the PowerPC

So, what is the solution for the troubled programmer? He should change the `int` variable `score`, and all other `int` variables, to an integral data type other than `int`. In this case, `score` could be changed to a `short` and the conditional test should be altered to include the macro in `limits.h` that defines the maximum value a `short` can have:

```
short  score;
. . .
. . .

if ( score > SHRT_MAX)
    Display_Number_Too_Big_Message();
```

Since the `short` data type is always a 2-byte type—regardless of the development environment—the code will compile the same regardless of whether it is compiled with a 680x0 compiler or a PowerPC compiler. Backwards-compatibility is maintained.

In summary, search for all occurrences of the `int` data type in all of the source code files of the project that is being ported. Then examine the context in which each `int` variable is used. Change each `int` variable to a different integral data type—such as the `short` type or the `long` data type. In any environment, these data types are 2-byte and 4-byte types, respectively.



The exception to the rule of changing a variable's type from `int` to a different integral type is when the variable will be used as an index. Indices are worked with (incremented, decremented, etc.) in registers. The CPU will work more efficiently with integers than it will with other data types. A loop index is one instance. Another is an array index. The address of an array member is calculated using the array index as an offset, so registers are involved.

Use ANSI Function Declarations

A C compiler always expects a function to return something. If no value is returned, then that function returns void:

```
void Post_Error_Alert( Str255 err_str )
{
    // function body
}
```

Since some C compilers will compile a function that doesn't explicitly state a return type, it may be tempting to declare the above function as follows:

```
Post_Error_Alert( Str255 err_str )
{
    // function body
}
```

Because the above declaration of `Post_Error_Alert()` lists no return type, it may appear that the function returns void. In fact, a function that is declared with no return type by default returns an `int`. Failing to include a return type makes the processor work a little harder than necessary. When the function has completed its execution, the processor goes to the effort of returning to the calling function whatever value results from the last statement in the called function. If you aren't concerned about the extra work the processor performs, you should be concerned that this type of function declaration can potentially introduce a bug into your program. Consider this trivial example:

```
My_Function( long num )
{
    num = 10;
}

void main( void )
{
    long test_num;
    long result;
```

Programming the PowerPC

```
test_num = 5;  
result   = 2;  
  
result = My_Function( test_num );  
}
```

In this example, the declaration for `My_Function()` lists no return type. After `My_Function()` has executed, what should the value of the `result` variable be? Even though `result` is given a value of 2 just before the call to `My_Function()`, after the function completes `result` will have a value of 10. Since `My_Function()` declares no return type, an `int` will be returned. And because the value of the last (and only) statement in `My_Function()` is 10, 10 is the value of the returned `int`. Figures 10.5 and 10.6 show the program output in the Metrowerks debugger.

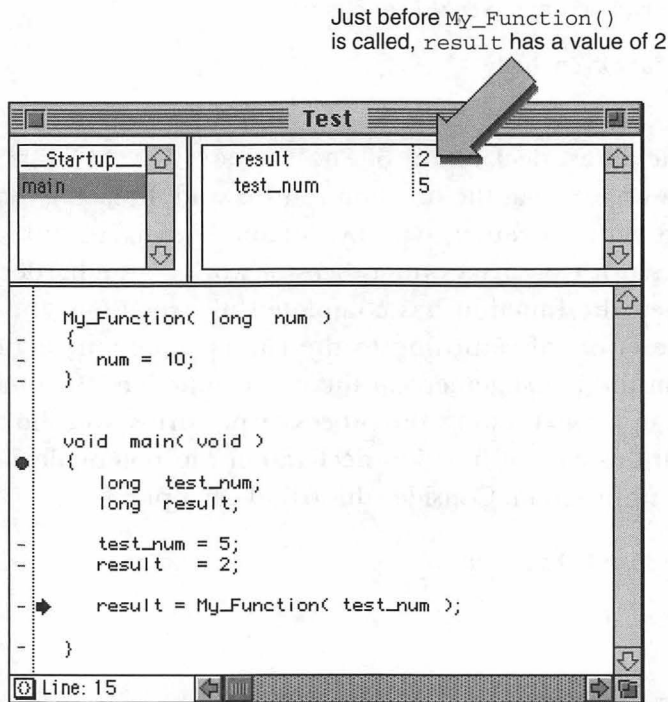


FIGURE 10.5 MONITORING THE RESULT VARIABLE BEFORE THE CALL TO `My_Function()`.

Chapter 10 Porting Code to Native PowerPC

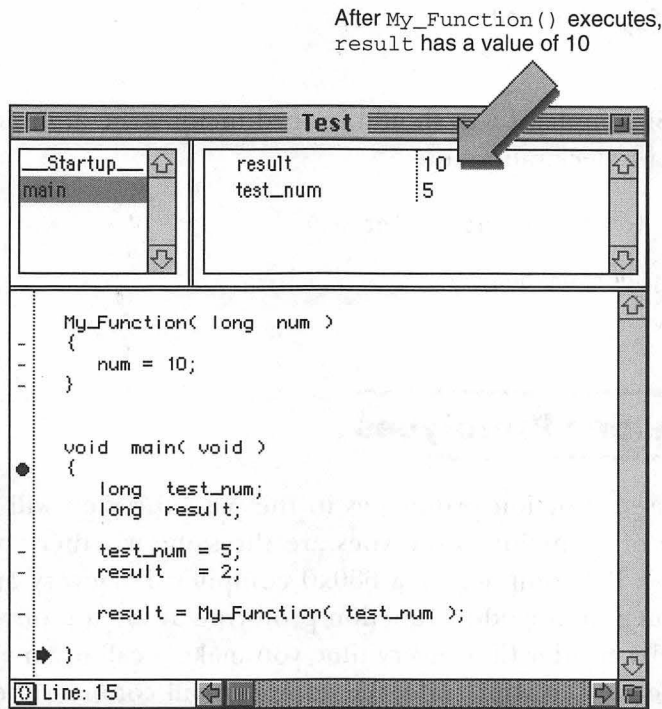


FIGURE 10.6 MONITORING THE RESULT VARIABLE AFTER THE CALL TO `My_Function()`.

The good habit of including function return types should carry over to function argument types as well. For functions that have no arguments, include `void` in the function declaration:

```
void Write_Warning( void );
```

If you're still using the pre-ANSI style of writing function declarations, you'll want to switch to the ANSI style. Before ANSI, function argument names were written in the function declaration, while the argument types were listed after the declaration:

```
int X_To_The_N( x, n )
int x, n;
{
```

Programming the PowerPC

```
    // function body
}
```

To be ANSI-compliant you should instead include the argument type in the function's declaration line:

```
int  X_To_The_N( int x, int n )
{
    // function body
}
```

Use Function Prototypes

If you've used function prototypes in the past, this step will already be taken care of. Function prototypes are the same whether you compile with a PowerPC compiler or a 680x0 compiler. For every application-defined function, include a function prototype at the top of your source code (or in a header file). Every time you make a call to an application-defined function in your code, the compiler will compare the data type of each passed parameter in the function call with the data type listed in the function's prototype. So while function prototypes are extremely easy to implement, they are also one of the most powerful tools you can use to quickly catch potential bugs.

When creating a function prototype you have the option of either including or omitting the names of the function's parameters. Consider this application-defined function:

```
long  X_To_The_N( long x, short n )
{
    int  i;
    long result = 1;

    for ( i = 0; i < n; i++ )
        result *= x;

    return ( result );
}
```

Chapter 10 Porting Code to Native PowerPC

The `X_To_The_N()` function can have a function prototype that looks like this:

```
long X_To_The_N( long x, short n );
```

At your discretion, the `X_To_The_N()` function could instead have a function prototype such as the following:

```
long X_To_The_N( long, short );
```



The choice of prototype styles is that of the programmer. The compiler has no preference. In this book you'll see function prototypes without the variable names.

USING A SINGLE SOURCE FILE FOR BOTH 68K AND POWERPC DEVELOPMENT

In Chapter 8 you saw that to develop a fat binary you must create two separate projects. One project is for 680x0 compilation, the other is for PowerPC compilation. The resulting executables can then be merged into one fat application. This process is the same whether you're working in the Symantec environment or the Metrowerks environment.

The fat application example in Chapter 8 had its own project and one source code file for compilation using a PowerPC compiler. A separate project and a copy of the source code file were then used for compilation with a 680x0 compiler. Two copies of the source code file were made because there are sometimes changes that have to be made in order for the source to compile on both a 680x0 compiler and a PowerPC compiler. In this section you'll see how you can use a single source code file—without modifications—with both compilers.

Allowing your code to be compilable with both 680x0 compilers and PowerPC compilers isn't just an aid to help you develop fat binaries from your new projects. It will also help you when you port existing 680x0

Programming the PowerPC

code to PowerPC code. Rather than just haphazardly make changes to existing code, you'll want to make sure your ported code still compiles with 680x0 compilers as well as with PowerPC compilers. That will allow you to keep your applications backwards compatible. You'll find it easy to create a 680x0 application, PowerPC-only application, and a fat binary—all from the same source code file.

Using Conditional Compilation Directives

Apple's universal header files exist to assist developers in their development of Macintosh applications—regardless of the compiler platform or platforms they use. As you saw in the discussions of universal procedure pointers in Chapter 7, conditional compilation directives play a big role in making the header files universal.

Since you're familiar with universal procedure pointers, I'll continue to use them in an example. Consider that the universal headers have an `#if` and `#else` preprocessor directive to force a compiler to use the proper definition of a `ModalFilterUPP`. A 680x0 compiler will define `ModalFilterUPP` to be a `ModalFilterProcPtr`—a `ProcPtr`. A PowerPC compiler, on the other hand, will define a macro called `USEROUTINEDESCRIPTORS`, which will cause `ModalFilterUPP` to take on a different definition. On a PowerPC compiler a `ModalFilterUPP` is defined to be a `UPP`:

```
#if USEROUTINEDESCRIPTORS

typedef UniversalProcPtr ModalFilterUPP;
. . .
. . .
#else

typedef ModalFilterProcPtr ModalFilterUPP;
. . .
. . .
#endif
```

By using the `#ifdef` conditional directive, the universal headers can force different definitions upon the same type. The above example shows

that the programmer need not be concerned with the actual type definition of `ModalFilterUPP`. Using the universal headers, the programmer's development environment will choose the proper definition for `ModalFilterUPP`—without assistance from the programmer.

The universal header files use conditional directives in hundreds of instances to greatly minimize the headaches of writing compiler-specific code. They don't, however, completely eliminate the need for you to include a few conditional directives in your own code.

QuickDraw Globals and Conditional Compilation Directives

Several global variables that most Mac programs make use of are QuickDraw global variables. A few examples are `thePort`, `screenBits`, and `randSeed`. The five standard patterns, `white`, `ltGray`, `gray`, `dkGray`, and `black`, are also QuickDraw global variables. While in the past it was acceptable to reference these variables directly, you must now preface their names with the name of the data structure which encapsulates them—`qd`. For example, if you called `InitGraf()` like this:

```
InitGraf( &thePort );
```

you must now call it like this:

```
InitGraf( &qd.thePort );
```

The reason for this is a change in the way the QuickDraw globals have been defined in the `QuickDraw.h` header file. In the older Apple `#includes`, the QuickDraw globals were defined to be in a struct named `qd`:

```
extern struct
{
    char    privates[76];
    long    randSeed;
    BitMap  screenBits;
    Cursor  arrow;
```

Programming the PowerPC

```
Pattern dkGray;
Pattern ltGray;
Pattern gray;
Pattern black;
Pattern white;
GrafPtr thePort;
} qd;
```

From the universal headers version of QuickDraw.h comes this new definition:

```
struct QDGlobals
{
    char    privates[76];
    long    randSeed;
    BitMap  screenBits;
    Cursor  arrow;
    Pattern dkGray;
    Pattern ltGray;
    Pattern gray;
    Pattern black;
    Pattern white;
    GrafPtr thePort;
};
```

You can see that in above struct definition varies from the older definition. Most significantly, no global qd variable is declared. If you're development environment doesn't declare such a variable for you, you'll have to do it yourself:

```
QDGlobals qd;
```



Does your development environment declare qd? Later in this chapter you'll see how to write conditional code that only makes the above declaration if the development environment doesn't already do so.

Regardless of where the qd variable is declared, you'll want to change all references to its members to include the qd preface. So a call such as this one:

```
InitGraf( &thePort );
```

should be changed to this:

```
InitGraf( &qd.thePort );
```

Though Apple's universal header files eliminate most of the 680x0/PowerPC compilation discrepancies, they don't handle all situations. One such case is the `QDGlobals` variable that I've been discussing. Though the following example doesn't apply directly to Metrowerks users, the general problem (and eventual solution) applies to users of any PowerPC compiler.

To successfully compile a PowerPC project using the Symantec cross development kit, you need to define a `QDGlobals` variable in your source code:

```
QDGlobals qd;
```

Before the PowerPC-based Macintosh, a global definition of `qd` was supplied by Apple in its runtime library—you did not have to declare this variable yourself. This situation is in fact the sole reason that the Symantec version of the Chapter 8 fat binary example used separate source code files for the two projects. The 680x0 file did not need to define the `qd` variable, while the PowerPC file did.

The solution to the problem of whether to declare or not declare `qd` involves another conditional compilation directive and another macro:

```
#ifdef __POWERPC
    QDGlobals qd;
#endif
```



NOTE

There are two underscores at the start of the `__POWERPC` macro.

In general, the `#ifdef` directive is written as:

```
#ifdef macro
```

Programming the PowerPC

```
// do something
#endif
```

Or, if more than one option is available:

```
#ifdef macro
    // do option 1
#else
    // do option 2
#endif
```

In this specific case the macro is `__POWERPC`. If you're compiling with a PowerPC compiler, `__POWERPC` is defined. That means the statement under the `#ifdef` will be compiled—and `qd` will be defined. If you're compiling with a 680x0 compiler, it turns out that the `__POWERPC` macro is not defined, and the declaration statement is thus skipped. This means that you can use the same source code file regardless of the compiler you use. Figure 10.7 illustrates how a PowerPC compiler will recognize the `powerc` macro and compile the `qd` declaration, while a 680x0 compiler working with the same source file will ignore the declaration.



Of course, you may still find it convenient to have two versions of the source code file. For the sake of being tidy, you might want to keep all the files for each version (680x0 and PowerPC) in their own respective folders. Then you'll have a version of the source code file in each folder. But, if you use the `#ifdef` and the `__POWERPC` macro, the two files will always be identical. And that's the real issue. Now it won't matter if you accidentally delete one file—the contents of the other is the same.

How the Compiler Knows If `powerc` Is Defined

Beside the `__POWERPC` macro, there are two other macros that can be used to determine if code is being compiled with a PowerPC compiler. The macros `powerc` and `__powerc` both have the same effect as `__POWERPC`. The `powerc` and `__powerc` macros are both used in many

Chapter 10 Porting Code to Native PowerPC

of the universal header files, and, like `__POWERPC`, they can be used in your own source code file. None of these macros are set in the universal header files, though. They each get defined by the compiler environment you use. This is best illustrated by looking at how the Symantec compilers work. Whether you're developing a 680x0 application or a PowerPC application, Symantec users begin by launching the THINK Project Manager. Once in the Project Manager, a developer can create a 680x0 project that includes a snippet like this one:

```
#ifdef __POWERPC
    QDGlobals qd;
#endif
```

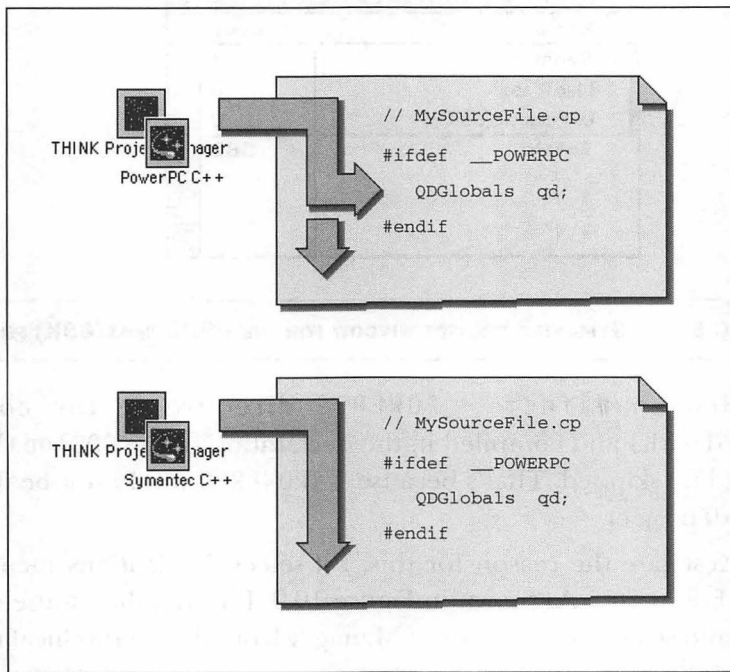
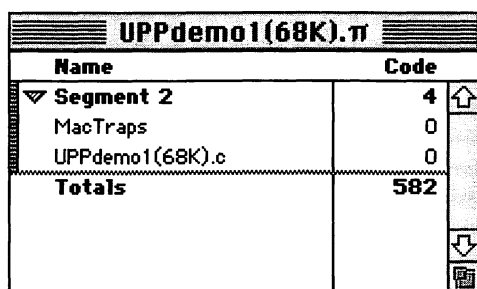


FIGURE 10.7 A POWERPC COMPILER WILL EXECUTE CODE UNDER AN `#IFDEF __POWERPC`; A 680x0 COMPILER WON'T.

Programming the PowerPC

The 680x0 project can be compiled, and, since it is a 680x0 project, `__POWERPC` will not be defined and the `QDGlobals` declaration will be skipped. Then, the project can be closed and a PowerPC project can be created. Using the same source code file, with the same snippet, the project can be compiled as PowerPC code. This time `__POWERPC` will be defined and the `QDGlobals` declaration will take place—as desired. This seemingly contradictory behavior can take place without ever leaving the THINK Project Manager, and without ever explicitly setting the `__POWERPC` macro.

To see how the Symantec environment handles the above situation, I'll open a 680x0 project, such as the `UPPDemo1(68K)` project I created in Chapter 8. This project window is shown in Figure 10.8.



Name	Code
▼ Segment 2	4
MacTraps	0
UPPdemo1(68K).c	0
Totals	582

FIGURE 10.8 THE SYMANTEC PROJECT WINDOW FOR THE UPPDemo1(68K) PROJECT.

If I added an `#ifdef __POWERPC` directive to the code of `UPPdemo1(68K)` and compiled it, the declaration of the `QDGlobals` variable would be skipped. That's because `__POWERPC` would not be defined for a 680x0 project.

To investigate the reason for this, I'll select the **Options** menu item from the Edit menu. As shown in Figure 10.9, I'll drag the mouse over to the right and select THINK Project Manager from the hierarchical menu.

In the dialog box that opens I'll choose to look at the Extensions options. Figure 10.10 shows the pop-up menu that will display that information.

In Figure 10.11 you can see that my C source code file will be compiled by the THINK Project Manager using the THINK C translator. The translators can be thought of as separate compilers that are run from

Chapter 10 Porting Code to Native PowerPC

within the THINK Project Manager. If my source code file was for a C++ program, it would have a .cpp extension and would be using the Symantec C++ translator.

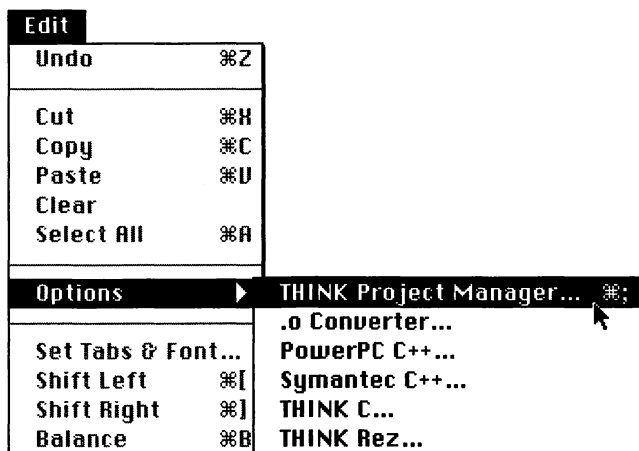


FIGURE 10.9 SELECTING THE OPTIONS DIALOG BOX FOR THE THINK PROJECT MANAGER.

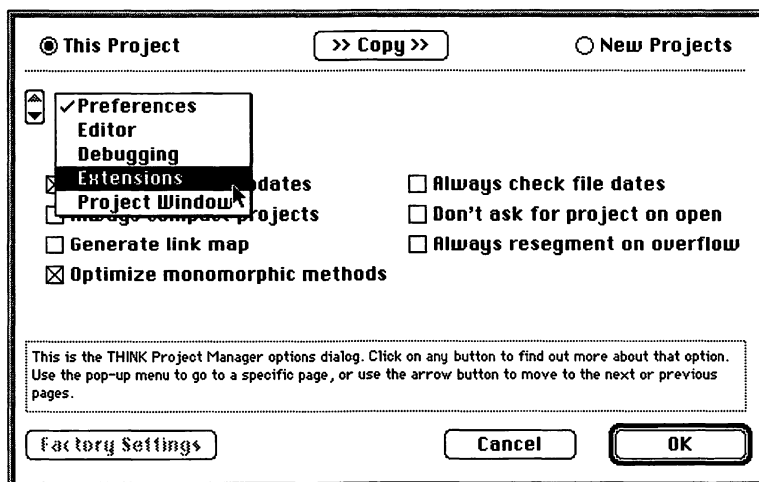
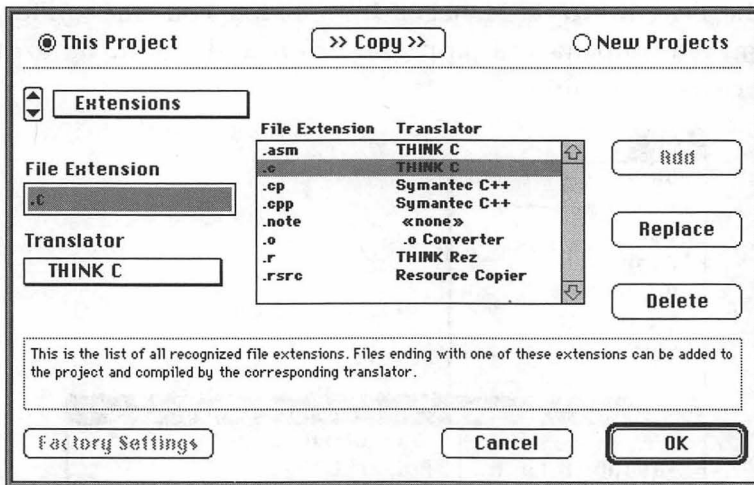


FIGURE 10.10 USING THE OPTION'S POP-UP MENU TO MOVE TO THE FILE EXTENSION INFORMATION SCREEN.

Programming the PowerPC



**FIGURE 10.11 THINK C FILE EXTENSION INFORMATION
IN SYMANTEC'S OPTIONS DIALOG BOX.**

After dismissing the options dialog box, and without leaving the THINK Project Manager, I'll close the 680x0 project and open a PowerPC project—such as UPPdemo1(PPC). The project window for this Chapter 8 project is shown in Figure 10.12. If I added an `#ifdef __POWERPC` conditional compilation directive to the source code file of this project and then compiled it, `__POWERPC` would be defined, and the line under it—the declaration of the `QDGlobals` variable `qd`—would be included in the compile.

With that project open, I'll again use the Options menu item as I did for the 680x0 project. The result is shown in Figure 10.13. Notice that for this PowerPC project, the THINK Project Manager is using a different translator—the PowerPC C++ translator.

The Symantec translators act as separate compilers—though they all run in the same THINK Project Manager environment. The code that defines `__POWERPC` is in the PowerPC ++ translator. If your project uses this translator—as PowerPC projects do—`__POWERPC` will be defined. If your projects use a different translator—such as the THINK C or Symantec C++ translator—`__POWERPC` will not be defined.

Chapter 10 Porting Code to Native PowerPC

UPPdemo1(PPC).π	
Name	Code
▼ Source	4
+ UPPdemo1(PPC).c	0
▼ Resources	4
cfrg.r	0
SIZE.r	0
UPPdemo1(PPC).rsrc	0
▼ Libraries	4
InterfaceLib.xcoff	0
MathLib.xcoff	0
PPC CPlusLib.o	0
PPCCRuntime.o	0
StdCLib.xcoff	0
StdCRuntime.o	0
▼ Scripts	4
PPCBuild.ts	0
Totals	594

FIGURE 10.12 THE SYMANTEC PROJECT WINDOW FOR THE UPPdemo1(PPC) PROJECT.

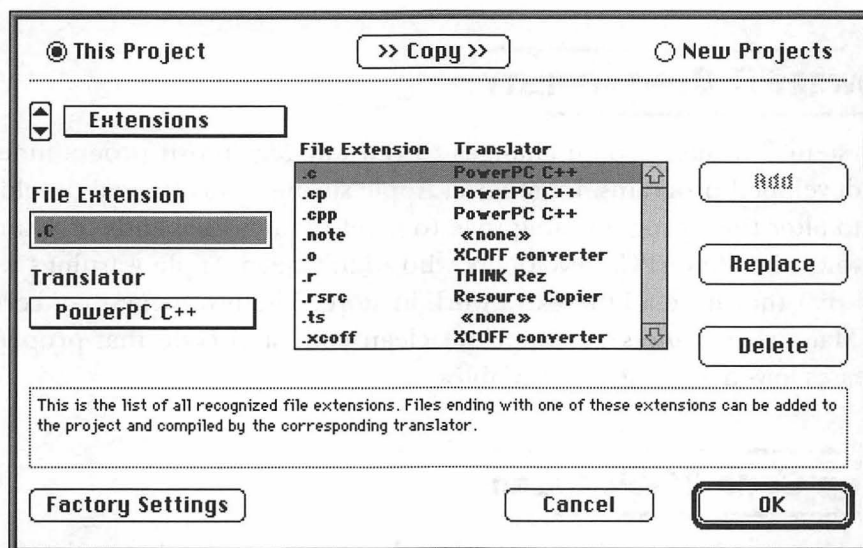


FIGURE 10.13 POWERPC FILE EXTENSION INFORMATION
IN SYMANTEC'S OPTIONS DIALOG BOX.

Programming the PowerPC

For the `powerc` and `__powerc` macros, Symantec does things just a little different. These two macros are defined in the PPC MacHeaders++ precompiled header file that gets included in each of your PowerPC projects. The net effect is the same—using `__POWERPC`, `powerc`, or `__powerc` in an `#ifdef` conditional directive will allow you to selectively choose which parts of your code get compiled—or don't get compiled. As you look at PowerPC code that was written by other programmers, you'll encounter one or more of these macros.

For the CodeWarrior compilers, things are just a little different. The Metrowerks PowerPC and 68K compilers are two separate applications. But the definition—or lack of definition—of `__POWERPC` works the same way. Launching the MW C/C++ PPC compiler sets the `__POWERPC` macro to be defined. Launching the MW C/C++ 68K compiler doesn't define it. Additionally, the `powerc` and `__powerc` macros are also defined by the compiler itself—not in the MacHeadersPPC precompiled header file. This differs from the Symantec compiler, which defines these last two macros in its PPC MacHeaders++ precompiled header.

POWERPC COMPATIBILITY

System 7 brought some changes to the way Macintosh programmers developed programs. In general, Apple strongly encouraged developers to alter their programming style to meet the new demands of System 7. Now, with PowerPC, developers who didn't heed Apple warnings will find that they have a little extra work in store. The two biggest concerns for Mac programmers will be 32-bit clean code and code that properly accesses low-memory global variables.

Keep Code 32-bit Clean

With the introduction of System 7 came the warning to developers that they should keep their code *32-bit clean*. Without modification, most existing applications were already 32-bit clean. Applications that weren't 32-bit clean generally did some fancy things with the upper 8 bits of 32-bit addresses.

Before System 7, the Mac used only the lower 24 bits of a 4 byte pointer. So some programmers used the upper 8 bits for application-specific purposes. When these applications were run on a Macintosh that considered all 32-bits of a pointer to be dedicated to an address, problems resulted.

Macs that are 680x0-based can, via the Memory control panel, switch addressing from 32-bits to 24-bits—even when running System 7. For the Power Macs, the 24-bit compatibility option that was provided for on 680x0 machines has been removed. Your PowerPC applications must be 32-bit clean.

How can you be sure your application is 32-bit clean? You can follow the guidelines as specified in the new *Inside Macintosh* books. Short of that, old-fashioned beta testing is still one of the best ways to make sure your application is 32-bit clean. Run your application on a Mac that has System 7 installed. If you're using a 680x0-based Macintosh, check the Memory control panel and verify that 32-bit addressing is turned on. If it isn't, turn it on and reboot your Mac. The Memory control panel is shown in Figure 10.14. If you're testing your application on a Power Mac, the Memory control panel—shown in Figure 10.15—won't have an option to toggle the number of bits in an address. The PowerPC will always use 32-bit addresses, so you're all set.

Use Access Functions for Low-Memory Globals

When an application is launched it receives its own section of RAM—its own *application partition*. The system also gets its own memory partition—the *system partition*. When a Mac boots up, system information is loaded into the system partition. While an application partition can start anywhere in a wide range of addressing space, the system partition always starts at the bottom of memory, at address 0.

The system partition consists of the *system heap* and, at the very bottom of the partition, a group of *system global variables*. Because of their physical position in memory, these system global variables are also referred to as *low-memory system global variables*, or *low-memory globals*. Figure 10.16 shows an overview of how the system global variables fit into the Mac memory scheme.

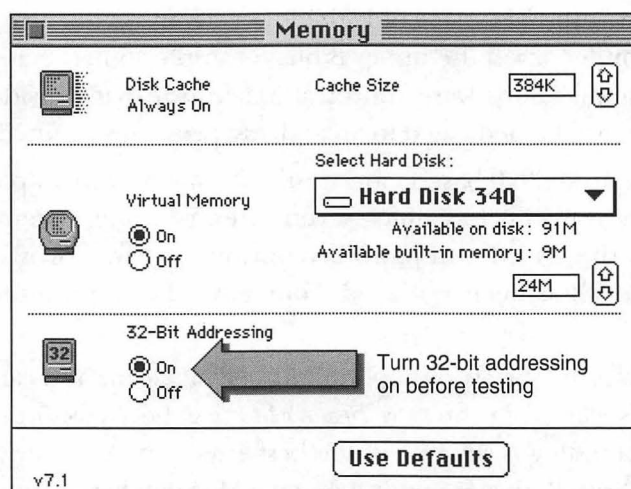


FIGURE 10.14 THE MEMORY CONTROL PANEL ON A 680x0-BASE MACINTOSH.

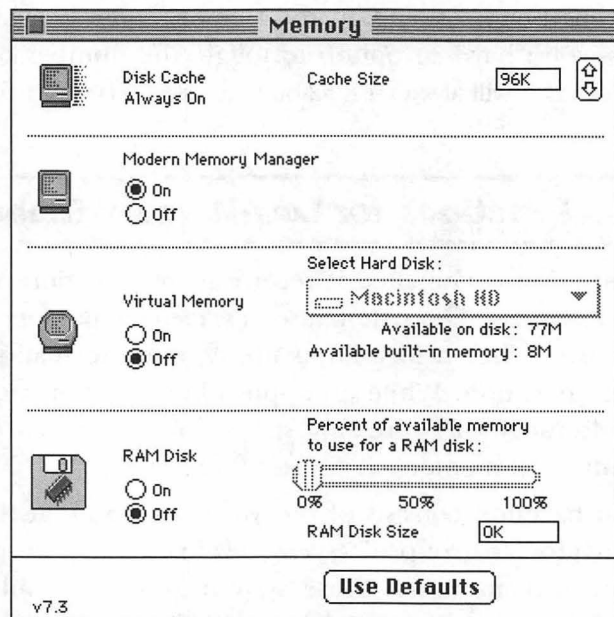


FIGURE 10.15 THE MEMORY CONTROL PANEL ON A POWER MACINTOSH.

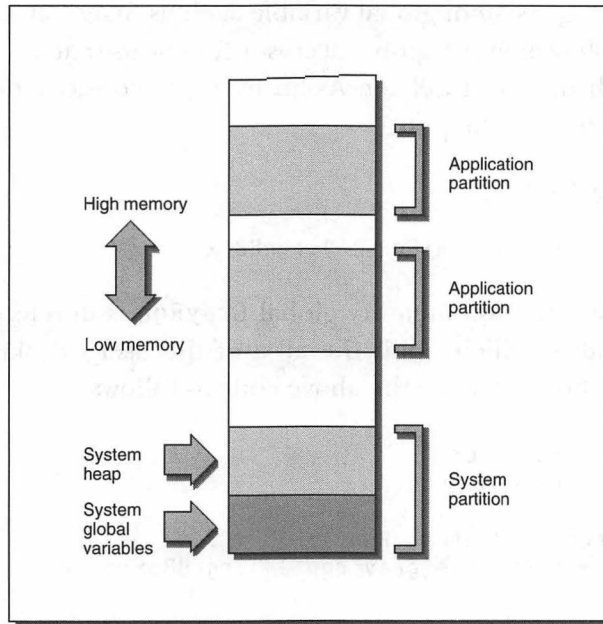


FIGURE 10.16 THE LOW-MEMORY SYSTEM GLOBALS RESIDE AT THE BOTTOM OF THE SYSTEM PARTITION.

The low-memory globals are used by the system to keep track of the operating system environment. One of the most commonly used low-memory globals is `GrayRgn`. If a programmer sets a window's dragging boundary to this region, that window will be able to be dragged about the entire desktop area of a user's machine—regardless of the size of the user's monitor or monitors.

In the past, Apple has stated that the best approach to using low-memory global variables involved *not* using them directly. While for now low-memory globals will always load to the same address each time your Mac boots up, Apple can't guarantee that the release of a new system version won't change the loading address of one or more low-memory globals. If that happens, code that relies directly on a low-memory global variable may have unpredictable results.

Apple's past recommendation has now become an unswerving rule—*don't* use a low-memory system global in your PowerPC code. Instead of

Programming the PowerPC

directly including a system global variable such as `GrayRgn` in your code, use one of the low-memory global accessor functions that are provided in the `LowMem.h` universal header. As an example consider how you may have used `GrayRgn` in the past:

```
Rect  Drag_Rect;

Drag_Rect = ( *( GrayRgn ) ).rgnBBox;
```

In the above snippet low-memory global `GrayRgn` is dereferenced twice to get at `rgnBBox`, which holds the area of the user's desktop. For the PowerPC, you should rewrite the above code as follows:

```
Rect      Drag_Rect;
RgnHandle the_gray_rgn;

the_gray_rgn = LMGetGrayRgn();
Drag_Rect = ( *( the_gray_rgn ) ).rgnBBox;
```

The low-memory global `GrayRgn` doesn't appear in the above code. Instead, the code relies on the Apple accessor function `LMGetGrayRgn()` to access `GrayRgn` and to return a handle to your program.

What happens if in the future Apple moves `GrayRgn` to a different address? Apple will modify `LMGetGrayRgn()` appropriately and provide a patch for the routine in the new system. You won't have to be aware of the change, and existing applications you've developed won't have to be recompiled. Instead, an existing application that calls `LMGetGrayRgn()` will execute the patched version of this routine. The patched version of this routine will know how to find `GrayRgn`. An application that relied on the global `GrayRgn` and didn't use the accessor function will not be able to locate `GrayRgn` and may fail.

The older Apple `#include` files consisted of an interface file named `SysEqu.h`. This file, which is not part of the new universal headers set of files, defined the memory addresses of the low-memory system global variables. Here is just a small part of the `SysEqu.h` file:

```
MainDevice = 0x8A4;    /* the main screen device      */
DeskPattern = 0xA3C;   /* Pattern desktop is painted */
```

Chapter 10 Porting Code to Native PowerPC

```
MBarHeight = 0xBAA; /* height of the menu bar */
GrayRgn = 0x9EE; /* Handle to desktop region */
```

You won't find the SysEqu.h header file anymore. If developers follow Apple's recommendation, there should never be a need for an application to know the address of a low-memory system global. Instead, rely on the accessor routines. To get the current value of any of the above four low-memory globals, you should use one of these accessor functions:

```
extern GDHandle LMGetMainDevice( void );

extern void LMGetDeskPattern( Pattern *DeskPatternValue );

extern short LMGetMBarHeight( void );

extern RgnHandle LMGetGrayRgn( void );
```

The LowMem.h header file also provides routines that allow you to change the value of any of the low-memory global variables. To set any of the same four globals listed above, you'd use these four accessor functions:

```
extern void LMSetMainDevice( GDHandle MainDeviceValue );

extern void LMSetDeskPattern( Pattern *DeskPatternValue );

extern void LMSetMBarHeight( short MBarHeightValue );

extern void LMSetGrayRgn( RgnHandle GrayRgnValue );
```

If your 680x0 project relies on any low-memory globals, search for an accessor function in the LowMem.h file found in the universal headers and instead use that accessor function.



If you've used the THINK C header file LoMem.h in the past, you'll want to now substitute the universal header file LowMem.h for it. That simply means changing the #include directive that appears at the top of your source code file or in your project's header file.

Use Universal Procedure Pointers in Place of ProcPtrs

Chapter 7 discussed universal procedure pointers at great length. Many of the porting suggestions provided in this chapter have been just that—suggestions. The use of UPPs, however, is mandatory. The failure to use `UniversalProcPtrs` when required will cause your PowerPC compiler to stop dead in its tracks.

Chapter 7 covered the theory and the use of UPPs. Here I provide a slightly different strategy for using them. Consider the following function from Chapter 7. It creates a routine descriptor for an application-defined routine called `My_Filter()`, opens a dialog box, and then disposes of the routine descriptor when the user is finished with the dialog box:

```
void  OpenModalDialog( void )
{
    // variable declarations
    ModalFilterUPP  my_filter_UPP;

    my_filter_UPP = NewModalFilterProc( My_Filter );

    // open dialog box

    ModalDialog( my_filter_UPP, &the_item );

    DisposeRoutineDescriptor( my_filter_UPP );

    // dispose of dialog box
}
```

The above example creates a local UPP, uses it, then disposes of it. A different strategy would be to create a global UPP variable rather than a local one. Then the routine descriptor is created once at program start-up, and remains in memory until the application quits. Using this scheme, there is no need to dispose of the routine descriptor—the system will perform that task when the program exits. Universal procedure pointers are nonrelocatable objects. If you want to declare one globally and have it remain in memory for the life of your program, you should allocate memory for it at the start of your program. That prevents it from

Chapter 10 Porting Code to Native PowerPC

occupying memory in the middle of your application's partition. Here's an example of how the filter procedure UPP could be used globally:

```
ModalFilterUPP My_Filter_UPP;    // declare globally

void main( void )
{
    // create routine descriptor at application startup

    My_Filter_UPP= NewModalFilterProc( My_Filter );

    // do stuff

    Open_Modal_Dialog();
}
```

With memory already allocated for the `My_Filter()` routine descriptor, `Open_Modal_Dialog()` doesn't have to create a local UPP variable or call `NewModalFilterProc()`. And when `Open_Modal_Dialog()` ends, the UPP shouldn't be disposed of—it will be used the next time `Open_Modal_Dialog()` is called. Here's a look at how `Open_Modal_Dialog()` would now look:

```
void Open_Modal_Dialog( void )
{
    // variable declarations

    // open dialog box

    ModalDialog( My_Filter_UPP, &the_item );

    // DON'T dispose of the routine descriptor here

    // dispose of dialog box
}
```

DATA ALIGNMENT

Compilers are designed such that they follow the data *alignment convention* of the processor they are compiling for. When a compiler places a data structure in memory, the compiler arranges the individual

Programming the PowerPC

elements that make up the data structure in a way that allows the target processor. When porting 680x0 code to native PowerPC code—and when writing PowerPC code that may run on a 680x0-based machine—you'll want to keep these differences in data alignment in mind.

The 680x0 Alignment Convention

When a data structure is placed in memory, its individual data members may not occupy contiguous bytes of memory. When compiling source code, a compiler may set up a data structure such that there is padding, or empty bytes, between some data members of a data structure. This is done to aid the processor in accessing individual data members. When padding is applied, and how much padding is applied, is subject to the processor the compiler is running on.

A 680x0 processor can access—that is, read or write—a word or a long word value at any even address. A single byte can be accessed at any address—even or odd. On a 680x0 processor a word is 2 bytes. That means the C data types `short` (2 bytes, or a word) and `long` (4 bytes, or a long word) can be aligned to any even address. The C data types `char`, which is a single byte, needs no alignment—it can appear at any address.

What happens when a data structure contains a number of different sized members, and by the nature of their ordering the start of some word and long word members don't fall on even addresses? The compiler supplies padding—empty bytes—to ensure that these members will in fact align to even addresses. Consider the definition of the following struct:

```
struct MyData
{
    char field_1;    // 1 byte
    long field_2;    // 4 bytes
};
```

When compiled on a 680x0 processor, a variable of the above struct type will occupy 6 bytes of memory—not the 5 bytes that the two data members require. Because the first member is only a single byte in size, a single byte

Chapter 10 Porting Code to Native PowerPC

of padding will be added to ensure that the second member—a long word—will begin on an even address. Figure 10.17 illustrates this.



Note that Figure 10.17 uses the Macintosh convention of showing a data object with its starting address as the larger address, and the object data members appearing below the starting address—down towards smaller addresses.

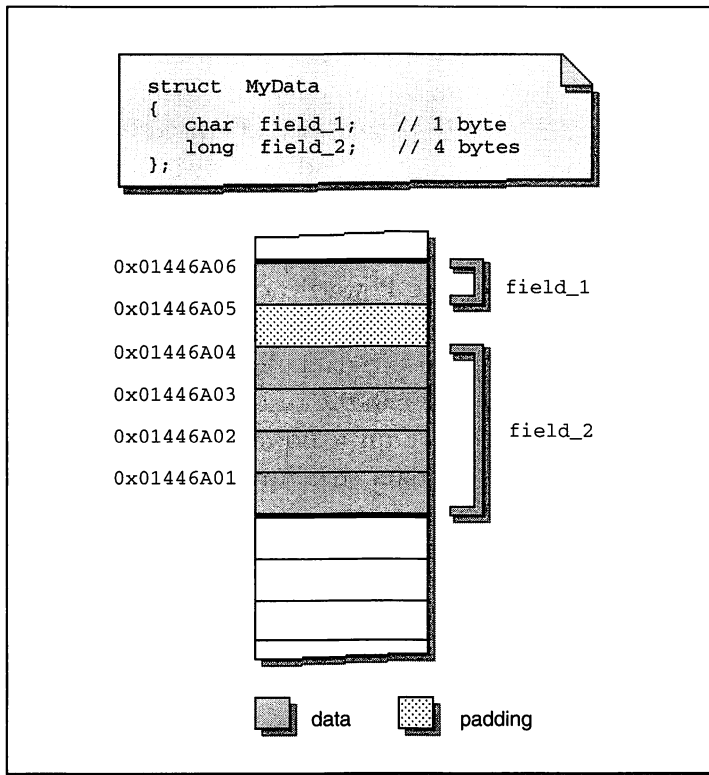


FIGURE 10.17 DATA ALIGNMENT OF A STRUCT ON A 680x0-BASE MACINTOSH.

After supplying padding to coerce individual struct members to fall on the appropriate boundaries, the 680x0 compiler will check to see what the overall size of the struct is. If it occupies an odd number of bytes,

Programming the PowerPC

the compiler will add a single byte of padding at the end of the structure to force it to occupy an even number of bytes.

The PowerPC Alignment Convention

Like a 680x0 compiler, a PowerPC compiler may add padding to a data structure. The alignment convention for the PowerPC processor, however, differs from that of the 680x0 processor.

A PowerPC can access data in exactly the same manner as a 680x0 processor—but it won't unless specifically requested to do so. Instead, accesses data that is aligned on a boundary that matches the size of the data. Data occupying 1 byte will be accessed at any address divisible by 1. Two byte data will be accessed at any address divisible by 2. Four byte data will be accessed at any address divisible by 4. The same pattern applies to data larger than 4 bytes.

Like the 680x0 compiler, the PowerPC compiler will insert padding into a data structure to force its individual members to fall on the above mentioned boundaries. Because the data alignment scheme is different for a PowerPC than it is for a 680x0, the resulting size the data structure occupies in memory may differ on a PowerPC-Mac than it will on a 680x0-base Mac. Consider the data structure used in the previous section's example:

```
struct  MyData
{
    char   field_1;        // 1 byte
    long   field_2;        // 4 bytes
};
```

When compiled on a 680x0 processor, a variable of the above struct type will occupy 6 bytes of memory. The 680x0 compiler inserts one byte of padding after the char field to force the long field to start on an even address. When compiled on a PowerPC processor, a variable of the above struct type will occupy 8 bytes of memory. The PowerPC alignment convention states that a long should start at an address divisible by 4, so the compiler inserts three bytes of padding after the first member. This is shown in Figure 10.18.

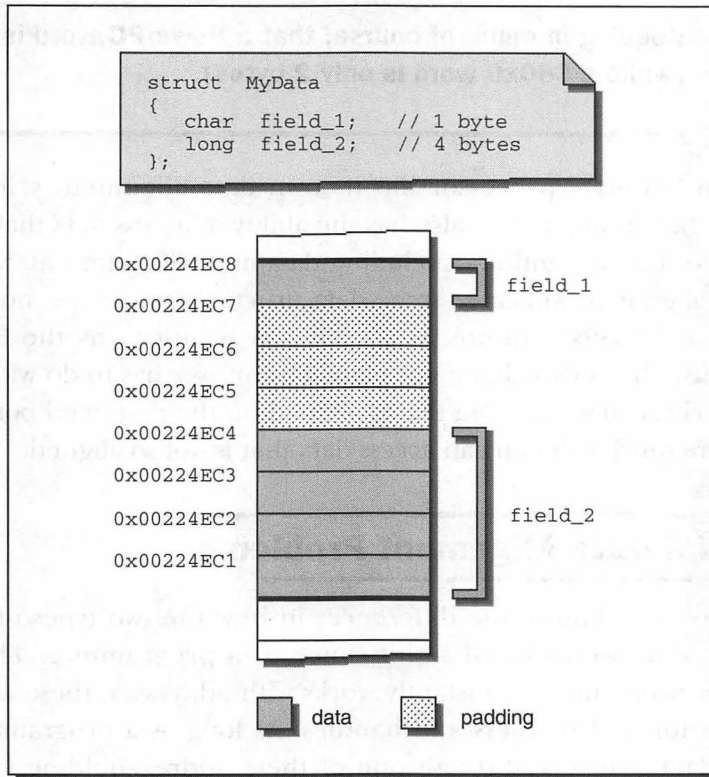


FIGURE 10.18 DATA ALIGNMENT OF A STRUCT ON A POWER MACINTOSH.

When the padding to the structure members is complete, the PowerPC compiler will add padding to the end of the structure in certain circumstances. If the entire structure is one or two bytes in size, no padding will be added. If the entire structure is greater than 2 bytes in size, and the structure doesn't end on a word boundary, padding will be added to the end of the structure to force the structure to end on a word boundary. For example, a 3 byte structure will receive 1 final byte of padding (to make a 4 byte structure), a 5 byte structure will receive 3 bytes of padding (to make an 8 byte structure), and a 9 byte structure will receive 3 bytes of padding (to make a 12 byte structure).

Programming the PowerPC



Keeping in mind, of course, that a PowerPC word is 4 bytes (while a 680x0 word is only 2 bytes).

While the PowerPC processor has its own data alignment scheme for accessing data in memory, it also has the ability to access data that doesn't conform to this convention—including data in a 680x0 format. Since the PowerPC alignment will cause some data structures to occupy more memory than their 680x0 counterparts, you may wonder why the PowerPC wouldn't use the 680x0 alignment plan. The answer has to do with speed. The PowerPC can access data that is aligned on the described boundaries much more quickly than it can access data that is not so aligned.

Potential Data Alignment Problems

Under most conditions, the differences in how the two types of processors align data won't be of significance to a programmer. Though a Macintosh programmer constantly works with addresses, these addresses are in the form of pointers and handles. As long as a programmer can access a data structure through one of these address-holding variables, the programmer need not be concerned with the exact address at which a data structure resides. There are some circumstances, however, when data alignment becomes a very important issue.

You need to be concerned about data alignment when there is the potential that your application will transfer data between a PowerPC and environment and a 680x0 environment. This can happen if your program is running on a PowerPC and the program creates a file that holds an application-defined data structure. If that file is transferred to a 680x0-based Mac, and that file is opened and read by a program running on a 680x0-based Macintosh, the file contents will not be properly loaded into memory.

The transfer of a data structure from one environment to another doesn't have to take place in the form of a physical medium for there to be potential alignment problems. If a network consists of both Power

Chapter 10 Porting Code to Native PowerPC

Macs and 680x0-based Macintosh computers, data transfer across the network can cause problems.

Rather than saying that data transfer from one Macintosh to another Macintosh causes problems, I chose to use the phrase “from one *environment* to another *environment*.” That’s because the transfer does not necessarily have to be between physical machines. If your PowerPC program passes a data structure to code that happens to be running under the 68LC040, that data structure will pass from the PowerPC environment to the 680x0 environment. Here the data transfer from one environment to another occurs within the confines of a single machine.

The Data Alignment Solution

Data alignment can be a problem. So, of course, compilers offer a solution. The solution comes in the form of the `#pragma` options directive. The `#pragma` directive is a C statement that is used to include compiler-specific information within a source code file. One of the many `#pragma` directives is the `#pragma options` directive.

The `#pragma options` directive is followed by an option name. The option name specifies which compiler setting you’d like changed. For data alignment, you’ll use the `align` option name.



You’ll find other `#pragma` directives in the documentation for the specific C or C++ compiler you use. While different compilers have different `#pragma` directives, both the Symantec and Metrowerks compilers support the `#pragma options` directives discussed here.

If you’d like to guarantee that a struct will be aligned such that both a Power Mac and a 680x0-based Macintosh can recognize it, use the `align=mac68k` option before the start of the structure definition. Then, immediately after the definition, use the `align=reset` option:

```
#pragma options align=mac68k
struct MyData
```

Programming the PowerPC

```
{  
    char field_1;    // 1 byte  
    long field_2;    // 4 bytes  
};  
#pragma options align=reset
```

Since the PowerPC alignment convention allows for quicker data access, you won't want to use the `align=mac68k` option throughout your entire source code file. Instead, you'll only want to mark particular struct definitions to be aligned in such a way. That's why you follow the struct definition with the `align=reset` option. This restores the alignment scheme to the PowerPC alignment convention.

The various `#pragma` directives will alter certain compiler settings. There's another way of changing compiler settings—through your compiler's preferences menu item. For instance, if you use the Metrowerks C/C++ PowerPC compiler you can globally set the alignment convention for a project from the Processor panel in the Preferences dialog box, as shown in Figure 10.19.

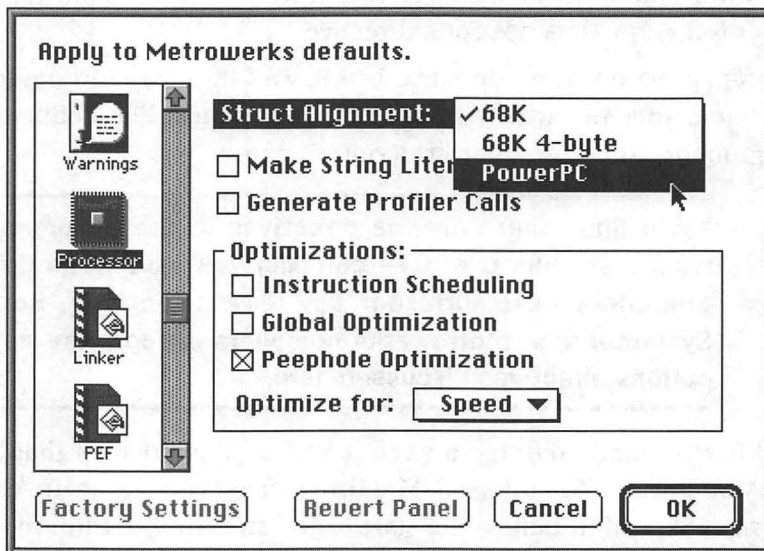


FIGURE 10.19 SETTING DATA ALIGNMENT IN THE METROWERKS COMPILER.

As mentioned, the preferences dialog box lets you choose struct alignment for the entire project. If you want to change the alignment for individual data structures, you'll use `#pragma` options directives in your source code.



You'll want to set the preference settings to PowerPC struct alignment for the entire project, then use `#pragma` options directives to mark individual data structures for 680x0 alignment. Whenever you use `#pragma` options `align=reset`, the compiler will return alignment to the type of alignment set in the preferences dialog box.

Testing Data Alignment

Since this entire discussion of alignment schemes sounds pretty theoretical, a few simple tests might be in order to verify it really is possible to switch back and forth between the different alignment schemes. First, consider this short C program:

```
struct  MyData
{
    char  field_1;
    long  field_2;
}

void  main( void )
{
    struct  MyData  the_data;

    the_data.field_1 = 0;

    printf( "Size of the_data = %d bytes", sizeof( the_data ) );
}
```

The above program defines a single structure and one variable of that struct type. It then uses a `printf()` call to write out the size of the struct variable.



The assignment statement that follows the variable declaration has no bearing on the size of the struct or on the outcome of the test. It is included for the sole purpose of preventing the compiler from optimizing the variable out of existence. If you declare a variable, but never use it, the compiler will remove it. This is an important fact to consider when setting up quick tests. If you ever can't "find" a variable in a debug window, it's because that variable was never used in any assignment statement.

What should the outcome of the `printf()` call be? If my compiler preference settings were set such that the project compiles code using the PowerPC alignment, the size of the `_data` should be 8 bytes. This is the same struct that was discussed several pages back, and pictured in Figure 10.18. Compiling the above code with the Metrowerks PowerPC compiler resulted in the output shown in Figure 10.20. As anticipated, the struct has a size of 8 bytes.

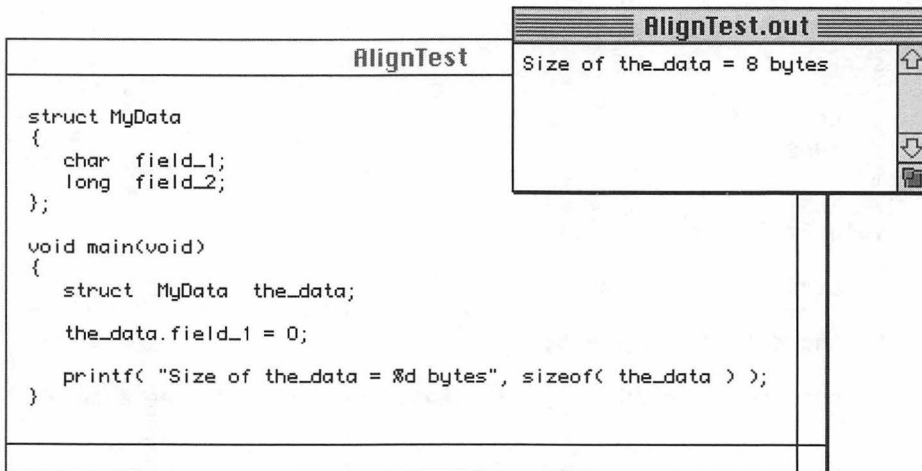


FIGURE 10.20 RESULTS OF TESTING DATA ALIGNMENT ON A POWER MACINTOSH.

Chapter 10 Porting Code to Native PowerPC



Once a structure is marked to be aligned by 680x0 data alignment conventions, any variable that is later declared to be of that struct type will appear in memory using the 680x0 alignment. You won't have to use the `#pragma options directives` on each individual variable.

To see if the `#pragma options` directive has the expected effect of altering the struct size in memory, I'll nest the struct definition between `#pragma` statements. The new version of the source code, and the output, are shown in Figure 10.21. Again, the results are as expected. The same structure now has a size of 6 bytes. You can refer back to Figure 10.17 to see how a 680x0-processor would align this structure in memory.

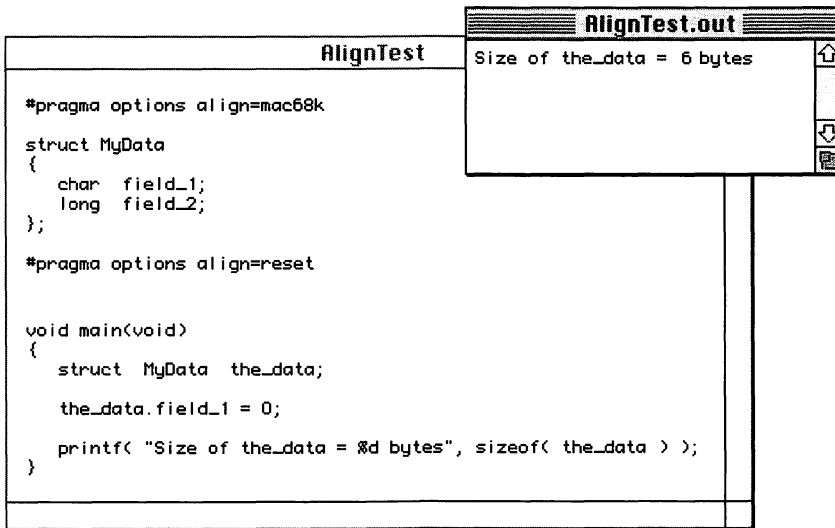


FIGURE 10.21 RESULTS OF USING #PRAGMA TO FORCE 680x0 DATA ALIGNMENT ON A POWER MACINTOSH.

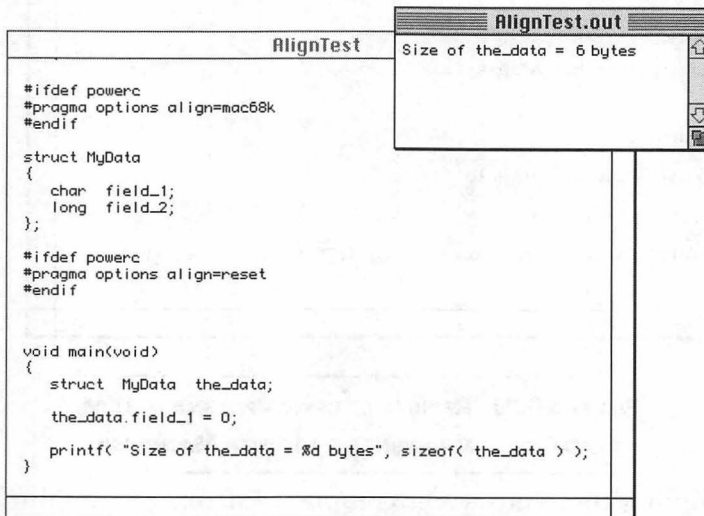
Before quitting the Metrowerks compiler, I'll run one final test. If this source code file might be used for compilation with both a 680x0 compiler and a PowerPC compiler, I should add a conditional test to see if the `#pragma` statement should be used. If I compile the source code with a 680x0 compiler, the data will be aligned using the 680x0 alignment con-

Programming the PowerPC

vention. There'll be no need to use a `#pragma options` statement. The way to perform this test is to use an `#ifdef` statement with the `powerc` macro that was described earlier in this chapter. Figure 10.22 shows the latest version of the `struct` code, along with the output. Because I compiled it using a PowerPC compiler, the `#pragma options` directive was compiled and the alignment of the `struct` was set to 680x0 alignment.



Here's today's quiz. If I used a PowerPC compiler (such as Metrowerks MW C/C++ PPC compiler), but ran it from a 680x0-based Macintosh, would the `#ifdef powerc` pass or fail? It would pass. The `#ifdef` isn't seeking to determine which Mac you are compiling on, but rather which Mac you're compiling for. A PowerPC compiler, regardless of the machine it is run on, generates an executable designed to run on a PowerPC. So, the `#ifdef powerc` is checking to see what type of compiler is being used, not what type of machine you happen to be running that compiler on.



**FIGURE 10.22 VERIFYING THAT THE POWERPC
COMPILER RECOGNIZES THE POWERC MACRO.**

Avoiding an Alignment Switch

A PowerPC processor cannot access data that is in the 680x0 data alignment format as quickly as it can access that same data if it is in the PowerPC data alignment format. So if it is at all possible, you'll want to manually rearrange your data structures so that they will appear in memory in both the PowerPC format and the 680x0 format. This statement might appear to be contradictory to previous discussions, so a short example is in order. Consider the following small struct:

```
struct MoreData
{
    short  field_1;    // 2 bytes
    long   field_2;    // 4 bytes
    short  field_3;    // 2 bytes
};
```

Though the above struct is 8 bytes in size, Figure 10.23 shows that a PowerPC compiler would align the `MoreData` struct such that it occupies 12 bytes. The last two bytes of padding are to force the structure to end on a word boundary, in compliance with the PowerPC data alignment guidelines.

A 680x0 will place the `MoreData` struct in only 8 bytes of memory. Because each member starts at an even address without the need for padding, none will be added between members. And because the overall size of the structure is an even number of bytes, no padding need be added to the end of the structure. This is shown in Figure 10.24.

By rearranging the members of the `MoreData` structure, it is possible to have the data structure align in the same way in both the PowerPC and the 680x0 environments. Here's how I've currently defined the `MoreData` structure:

```
struct MoreData
{
    short  field_1;    // 2 bytes
    long   field_2;    // 4 bytes
    short  field_3;    // 2 bytes
};
```

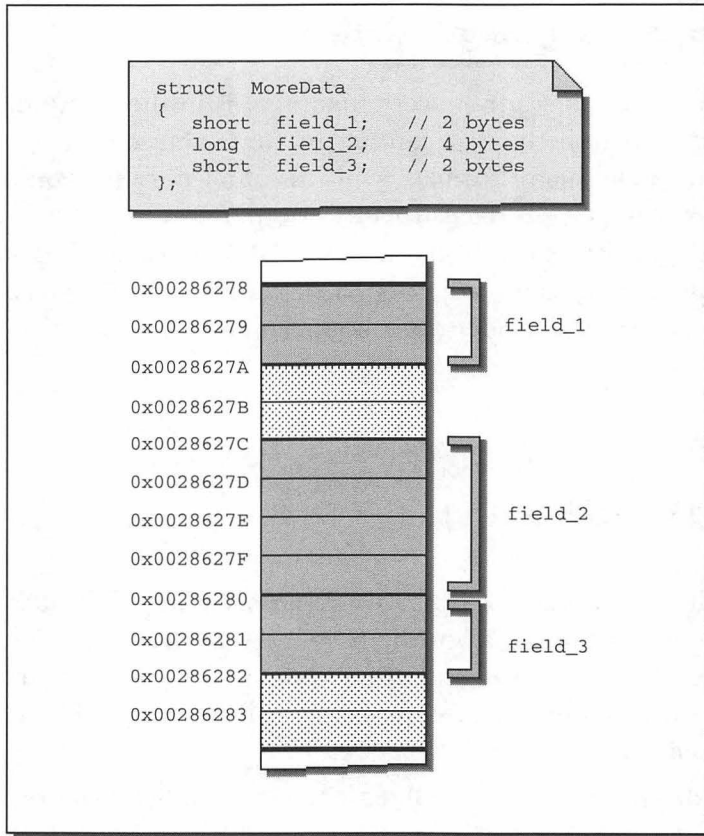


FIGURE 10.23 DATA ALIGNMENT ON A POWER MACINTOSH.

Since the order in which the data members appear in the structure is unimportant, rearranging them won't have an impact on their use later in the source code. Here I've simply switched the order of the `field_2` and `field_3` members:

```
struct MoreData
{
    short  field_1;    // 2 bytes
    short  field_3;    // 2 bytes
    long   field_2;    // 4 bytes
};
```

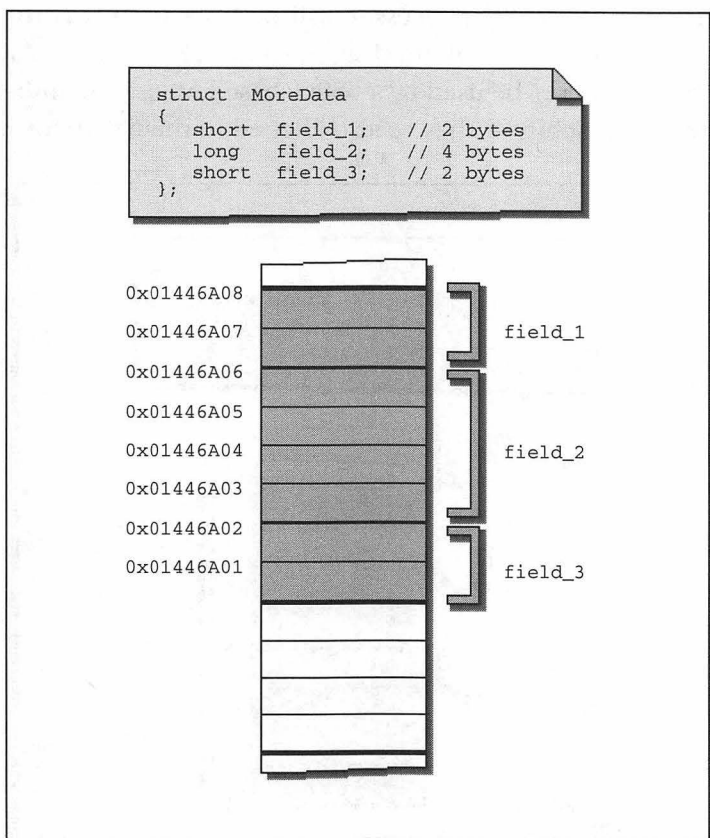



FIGURE 10.24 DATA ALIGNMENT ON A 680x0-BASED MACINTOSH.

Figure 10.25 shows how the `MoreData` structure will now be aligned in memory if the code was compiled with a PowerPC compiler. Notice that there is now no padding, and that the structure occupies the same amount of memory (8 bytes) as it would if it had been compiled using a 680x0 compiler.

For a struct that holds standard data types, an examination and manual rearrangement of its members can be done with relatively little effort. If the struct members are rearranged properly, no `#pragma options align` statement will be necessary. When an application is built

Programming the PowerPC

and executed, a PowerPC processor will be able to access the struct members quickly—they're aligned in the PowerPC format. And, if the data structure is to ever be used by a 680x0-based Mac, its members will be readable by that machine—they're also aligned in the 680x0 format.

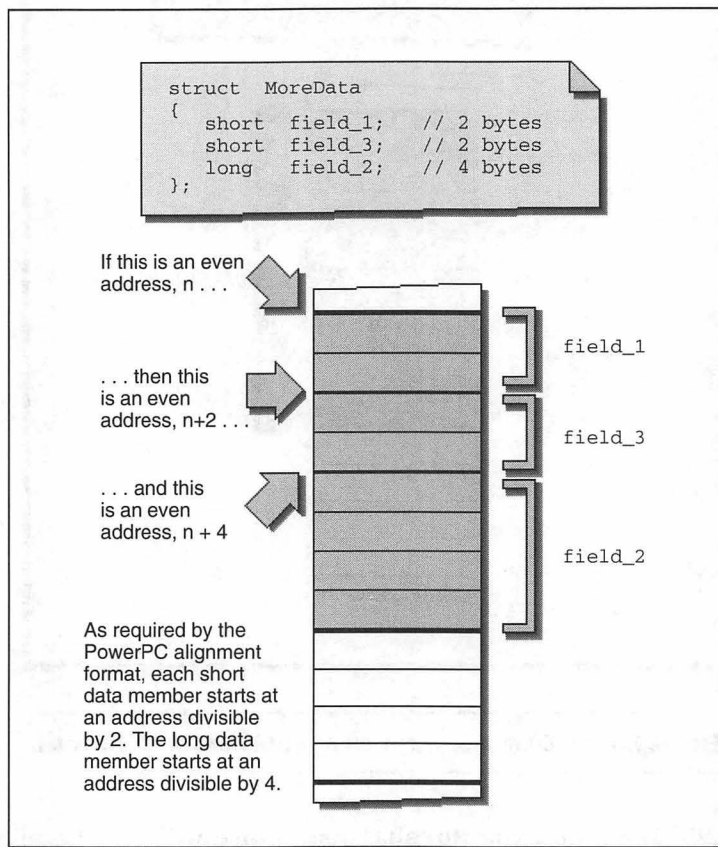


FIGURE 10.25 MANUALLY REARRANGING STRUCT MEMBERS TO FORCE DATA ALIGNMENT TO BE THE SAME ON BOTH A 680X0-BASE MACINTOSH AND A POWER MACINTOSH.

For larger structures, this manual rearrangement will not be worth the work. This is especially true of large structures that contain members whose size isn't readily apparent. For example, if you define a struct that has a `WindowRecord` as one of its members you might not want to

expend the effort necessary to determine where the `WindowRecord` and the other struct members should be positioned. For cases such as this you'll want to stick with the method of using the `#pragma options align directive`.

CHAPTER SUMMARY

While just about every program that runs on a 680x0 Mac with System 7 will also run on a Power Mac, these programs won't take advantage of the increased speed of the PowerPC microprocessor. Unless they are first ported to native PowerPC code.

The primary step in turning 680x0 code into native PowerPC code is to recompile the old code using one of the new PowerPC compilers. This book has covered two such compilers—the Symantec CDK and the Metrowerks CodeWarrior compilers.

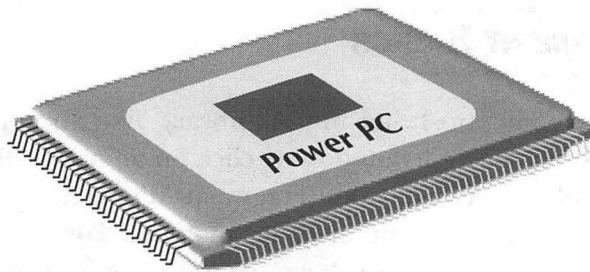
Before recompiling 680x0 code, you'll want to take a few measures to ensure that the compilation goes smoothly. First and foremost, you'll want to make sure that you're using Apple's universal header files. You'll also want to remove assembly code and replace it with C or C++ code. In almost all cases, the execution time of your code will not suffer—the speed of the PowerPC processor and the optimization algorithms of the PowerPC compilers will see to that. Some other porting recommendations from Apple are:

- Change `int` variables to variables of type `short` or `long`.
- Always list a function return type in a function declaration.
- Always use function prototypes.
- Keep your code 32-bit clean.
- Use access functions to examine or set low-memory system global variables.
- Use `UniversalProcPtrs` in place of `ProcPtrs`.

Programming the PowerPC

If you want to use a source code file as part of both a 680x0 project and a PowerPC project, you'll want to make use of conditional directives such as `#ifdef`. The `powerc` macro can be used to force the compiler to include or omit certain lines of code from compilation.

Compilers align data structures in memory. That is, they add padding between some data members of a structure in order to coerce those data members to start on certain address boundaries. The data alignment scheme used by the Power Macintosh is different than that used by the 680x0-based Macintosh. If your PowerPC program will write data structures to files that may be read by a 680x0 machine, or vice versa, you'll want to force the PowerPC compiler to use the 680x0 data alignment convention. You can do that by using the `#pragma options align=mac68k` directive.



CHAPTER 11

IMPORT LIBRARIES

Import libraries, or shared libraries, are a very powerful, yet easy, means of packaging code such that it can be shared by several applications. Import libraries are also a convenient way for you to implement “plug-in tools” capabilities to any of your PowerPC applications. In this chapter you’ll see how the code in an application loads the code in an import library, and then executes it. You’ll see how the Metrowerks PowerPC compiler makes it easy to turn a project into a shared library rather than an application.

Import libraries are powerful in their own right. When combined with an application that uses Apple Events, they become even more potent. This chapter sets the foundation for creating import libraries, and applications that load them. The next chapter shows how Apple Events can easily be added to an application to further enhance the usefulness of import libraries.

CODE FRAGMENT BASICS

On a Power Macintosh, an import library, like an application, is a code fragment. When you double-click on an application, the operating system will invoke the Code Fragment Manager routines that are necessary to load the application code fragment. For an import library code fragment, however, you'll have to intervene and make the Code Fragment Manager call yourself. So, before delving into import library code, a little background information on code fragments is in order.

All Code is a Fragment

On the PowerPC, a code fragment is any block of executable code. Whatever code you write must eventually be turned into a fragment. Whether your fragment is standalone code, or code that relies on that found in another fragment, your PowerPC compiler will take care of the details of marking code as a fragment.



Of course, it's the linker that is the part of your development environment that actually creates a fragment. An environment's editor, compiler, and linker have become so integrated, though, that people generally refer to individual actions such as editing, compiling, or building, as being performed by "the compiler."

On the 680x0-based Macintosh, code fell into clearly defined categories. On the Power Mac, the boundaries between types of code have become blurred. Though a fragment can be considered to fall into one of a few categories, the distinction between fragment types is trivial. From Chapter 5 you may recall that a code fragment is usually thought of as being in one of the following categories:

- An *application* code fragment. Like a 680x0 application, a PowerPC application is standalone code that needs no other code

(aside from system software) to execute. It can optionally, however, make use of the code in other fragments.

- An *import library* code fragment. This fragment type, which is also called a shared library or dynamically linked library, holds code and data that are to be accessed by one or more other fragments. Unlike an application fragment, an import library requires at least one other fragment to call the routines it holds. Often this other fragment is an application fragment.
- A *code resource* code fragment. A code resource holds executable resource code. A control definition, or CDEF, is an example.
- An *extension* code fragment. This fragment type adds to the capabilities of other fragments. QuickTime is the most notable example of an extension fragment.

Fragment Code and Containers

In keeping with the Power Mac generalization that any executable code is a fragment, all PowerPC code fragments are thought to be kept in *containers*. The container is the physical holding area of executable code. Just what a particular container *is* can vary widely from fragment to fragment. In general, if the Mac operating system can access it, it might be a container.

Because PowerPC system software now exists as an import library, it resides in a container. So one type of container is the ROM of a Power Macintosh. The import libraries that you'll create in this chapter will be stored in files. One such import library file type is the 'shlb' file—so this file can be thought of as a container holding executable import library code. An application fragment's data fork is still another type of container. For a PowerPC application, the data fork holds the application's executable code—Figure 11.1 is a modification of a Chapter 8 figure, and serves as a reminder of how a PowerPC application looks.

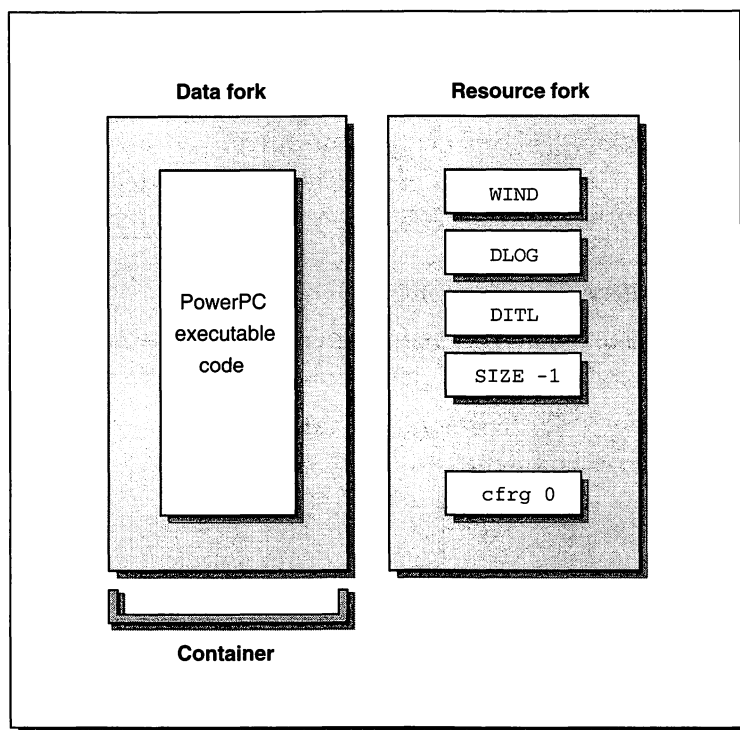


FIGURE 11.1 THE DATA FORK OF A POWERPC APPLICATION HOLDS THE APPLICATION'S EXECUTABLE CODE, AND IS CONSIDERED A CONTAINER.



Up to this point, all of the example code in this book has been for source code that gets turned into applications—files of type 'APPL.' So, perhaps without being aware of it, you've already worked extensively with fragments and containers. Each fragment, however, has had the same general purpose—to serve as a standalone application. In this chapter you'll be exposed to fragments with other purposes.

Before a fragment's executable code can be used, it needs to be loaded into memory. For an application fragment, this act is performed by the Code Fragment Manager when the user double-clicks on the applica-

tion's icon. For a different type of fragment, you, the developer, will have to include a call to a Code Fragment Manager routine.

The different kinds of fragments fall into one of two different formats that the Code Fragment Manager is capable of recognizing—the Extended Common Object File Format (XCOFF) and the Preferred Executable Format (PEF).

XCOFF is a format derived from the Common Object File Format (COFF). XCOFF is an IBM format used primarily on UNIX-based computers. The format that Apple has defined and uses, and more fully supports on the Power Mac, is PEF. Apple has improved upon XCOFF by creating a container format that occupies much less space. Besides the obvious advantage of saving on hard disk real estate, the smaller size makes it easier and faster for the Code Fragment Manager to load a PEF container than an XCOFF container.



So why does Apple bother to support XCOFF, even to a limited extent? IBM, Motorola, and Apple all participated in the development of the PowerPC chip. The original development tools generated code in the IBM XCOFF format. As time goes on, Apple support of XCOFF will diminish.

IMPORT LIBRARY BASICS

On a Power Macintosh, both an application and an import library are code fragments. The executable code of an application is stored in the data fork of a file. While the executable code of an import library may be stored in ROM or in a resource, you'll typically use a file as its container. The primary distinction between an application and an import library is that an application is double-clickable code, while an import library cannot execute without the support of another fragment—often the application fragment.

Imported and Exported Symbols

All fragments contain *symbols*. A symbol is used as a reference point that allows code—either code in the fragment or in a different fragment—to access a routine or variable. A symbol that is referenced by code in a different fragment is called an *exported symbol*, or *export*.

An import library fragment contains exported symbols—symbols that will be referenced by the fragment or fragments that use the library code. These other fragments contain *imported symbols*, or *imports*, that will eventually correspond to the exported symbols.

When a fragment is built by your development environment, an imported symbol is created for each reference to external code. Since the memory location of code isn't known until runtime, when it is loaded, the symbols must suffice during the build process. Then, each time a fragment is loaded into memory (such as when an application is launched), the Code Fragment Manager replaces all imported symbols with memory addresses. These addresses are the addresses of referenced routines in external code.

Even upon the launching of an application, the Code Fragment Manager may not be able to resolve all symbols in that application fragment. This can occur when an import library used by the application isn't already loaded in memory. If the library is not marked by the application to be loaded at application launch, the Code Fragment Manager will wait until the import library is eventually loaded before resolving these symbols.



When an application launches, why wouldn't it load any import libraries it uses? Because a library may contain seldom used code that gets executed only when a particular menu item is selected. You'll see a specific example of this later in this chapter.

When the Code Fragment Manager finally does resolve the import symbols in a fragment, it creates a *connection* to the fragment that holds the

referenced code. When using import libraries, you'll use the *connection ID* that the Code Fragment Manager returns to your application. This connection ID can be used by your application to sever the connection and unload the fragment code when your application is through with it.

Import Library Special Routines

A shared library fragment creates a list of symbols that it exports to fragments that will be importing the shared library's code. This list of export symbols is mandatory and necessary—without it, a fragment wouldn't have access to the variables and functions in the shared library it was trying to load and execute. Besides this mandatory list of symbols, a shared library can optionally define three special symbols. Each symbol represents a function in the shared library, and allows other fragments access to these three functions. For that reason the three special routines are also called *entry points*. Two of these symbols allow two of the three functions to be invoked implicitly by other fragments. That is, the code within another fragment will be able to trigger the execution of one of these functions without the programmer ever including a call to it.



While a very interesting concept, this implicit function calling shouldn't be new to you. Dialog box user items are updated through this implicit type of function call too. Chapter 7 covers this concept in some depth.

The three special routines consist of an initialization routine, a main routine, and a termination routine. As mentioned, each of these special routines are optional.

If a fragment has an *initialization routine*, it will be executed by the application upon loading the import library. The application source code need make no explicit call to the initialization routine—it will be called just as soon as the import fragment is loaded. Regardless of how long the imported fragment remains in memory, this is the only time that the initialization routine will be executed.

Programming the PowerPC

When the initialization routine is called, it receives a pointer to a data structure that holds information about the import fragment's container. Since the application is loading the fragment, it should be obvious that the application has information about the fragment container. The initialization routine of the import library can use this information about its own container to do things such as access its own resource fork. In Figure 11.2, an application is loading an import library. Because the import library has defined an initialization routine, the application invokes it—passing it information about the import library container. In this example the initialization routine uses this information to find its own resource fork to access an 'ALRT' resource and then display an alert.

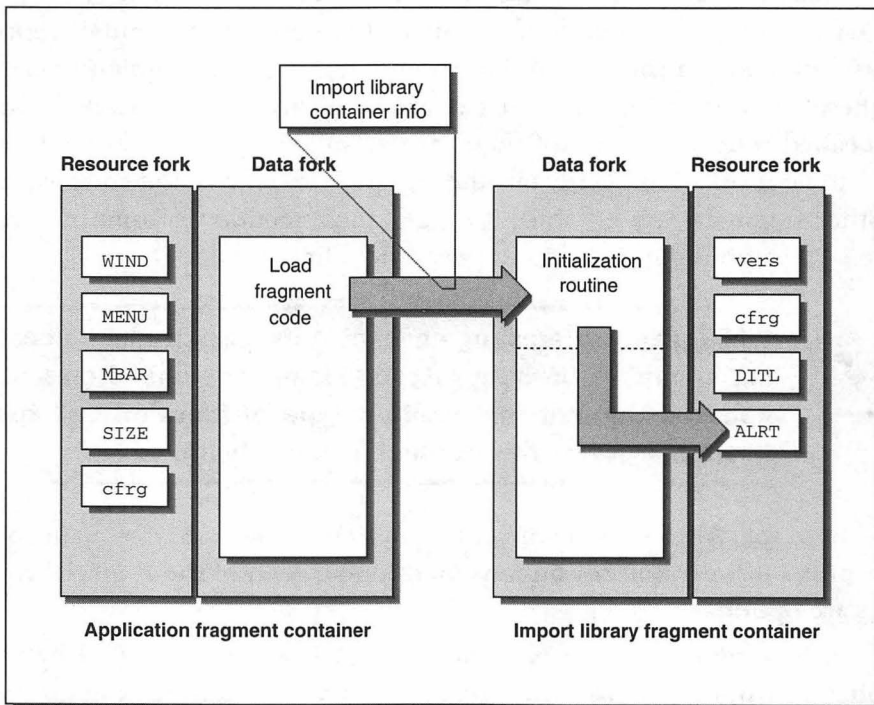


FIGURE 11.2 THE LOADING OF AN IMPORT LIBRARY TRIGGERS THE EXECUTION OF THE LIBRARY'S INITIALIZATION ROUTINE.



Later in this chapter you'll see complete source code listings for examples that do just what's shown in Figure 11.2.

When an application is finished with an import library it may keep the import library in memory for later use or unload it to free up memory. If the import library has a *termination routine* defined, that routine will automatically be invoked if the import library fragment gets unloaded. If a fragment defines a termination routine, its purpose is often to free memory that was allocated in the fragment's initialization routine.

The third optional special routine is a fragment's *main routine*. This routine has no one set purpose—though it often holds code that updates drawing that was performed by the fragment. This routine is also the only one of the three special routines that is not called automatically.

Consider the following hypothetical example. An import fragment displays the standard get file dialog box to allow the user to select a 'PICT' file, and then opens and displays the contents of that file. Since the posting of the standard get file dialog box only occurs once, that task could be performed by the fragment's initialization routine. This routine would also reserve the memory needed to hold the data from the opened 'PICT' file. The fragment's main routine would then draw the picture to a window. Since the main routine handles the drawing, it would be responsible for updating the window that holds the picture. Thus this routine would be called by the application whenever the window had to respond to an update event. When the application was finished with the picture, the import library code could be unloaded. As part of this unloading process the import fragment's termination routine would be invoked to release the memory that had been allocated for the picture.



The above example isn't entirely hypothetical. Chapter 12 has the source code for an example that has an import library that defines the three special routines as described above.

IMPORT LIBRARY CODE

On a PowerPC, any executable code is considered to be a code fragment. An import library is a code fragment that doesn't stand on its own—it relies on either an application fragment or another import library fragment to load and execute its code. This chapter's discussions—as well as all of the material in Chapter 5—have familiarized you with the theory behind fragments, containers, and import libraries. Now it's time to see some working code that backs up that theory.

This section describes the code you'll need to write to create a simple import library fragment. Later in this chapter, you'll see how to write the code for an application fragment that loads this import library. Finally, this chapter will show the CodeWarrior project files you'll need to create in order to compile and link the example code. The result will be two separate fragment files—an import library and an application that uses the import library.



While it is very possible for a shared library to import code from still a different shared library, all of the discussion in this section will assume an application fragment is importing code from an import library. That should minimize confusion by making it safe for you to assume that discussions of import symbols pertain to the application (it imports code from the library) and discussions of export symbols pertain to the shared library (it exports code to the application fragment).

Defining One of the Special Routines

Earlier in this chapter you read that an import library can define up to three special routines that serve as entrance points into the fragment. Here you'll see the specific code you'll need to write to create an import library that defines an initialization routine.

An import library special routine can have any valid function name. The function return type and argument, however, should follow the form shown here:

```
OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )
```

The initialization routine is called when the import library is loaded. When that happens, the application will pass a variable of type `InitBlockPtr` to the routine. The invocation, the filling of the data structure pointed to by the `InitBlockPtr`, and the passing of that pointer to the initialization routine are all done automatically when the Code Fragment Manager loads the library. Depending on the tasks the initialization routine is to perform, it may or may not make use of the import fragment container data that the `InitBlockPtr` points to. Consider this very trivial initialization routine:

```
OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )
{
    DrawString( "\pLibrary loaded." );

    return ( noErr );
}
```

The above function will get called when the import library of which it is a part of gets loaded. Its only task is to draw a string that gives notification that the library was loaded. By convention, the initialization routine returns an `OSErr`. In the above example, you can see that I've made the bold assumption that the `DrawString()` call didn't fail—I simply return the Apple constant `noErr`.

The Apple universal header file `FragLoad.h` defines the functions and data structures (including the `InitBlock` struct) you'll be using when you write import library code. Any source file that will be compiled into an import library needs to include it:

```
#include <FragLoad.h>
```

The initialization function that you've just seen makes several assumptions about the application that invokes it:

- A window is open.
- The port has been set to that window.

Programming the PowerPC

- The graphics pen has been appropriately positioned.

These assumptions are too important to make in “real” Macintosh programming. But making them here allowed me to demonstrate that an initialization routine is not difficult to write.

A Second Initialization Routine Example

The previous example demonstrated what’s required when writing an initialization routine—a function that has an `InitBlockPtr` as a parameter and an `OSErr` as its return type. Since the previous example was too trivial to be of any practical use, I’ll present a more usable initialization routine example here:

```
#include <FragLoad.h>

OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )
{
    FSSpecPtr the_FSSpec_ptr;
    short      res_ref_num;

    the_FSSpec_ptr = init_block_ptr->fragLocator.u.onDisk.fileSpec;
    res_ref_num = FSpOpenResFile( the_FSSpec_ptr, fsCurPerm );
    UseResFile( res_ref_num );

    Alert( 128, nil );

    CloseResFile( res_ref_num );

    return ( noErr );
}
```

When called, the above initialization routine will display an alert. Normally, the display of an alert is handled simply by calling the Toolbox routine `Alert()`. The initialization routine requires a little extra code because the alert that it displays is found in the resource fork of the import library itself—not in the resource fork of the application that loads the library. By keeping the alert resources (an ‘ALRT’ and a

'DITL') in resource fork of the import library, the library remains self-contained. Way back in Figure 11.2 you saw that an import library can have its own set of resources. That figure also illustrated how the import library fragment container information that is passed to the initialization routine can be used to direct the code found in the import libraries data fork to the resources found in its resource fork. You now know that this fragment container information comes in the form of an `InitBlock` data structure.

The `InitBlock` data structure consists of nine fields, one of which is another structure—this one of type `FragmentLocator`. Deeply embedded in the `FragmentLocator` structure is an `FSSpecPtr` named `fileSpec`. This pointer to an `FSSpec` can be used to gain access to the resource fork of an import library.



If you must know, the entire path to the `FSSpecPtr` is as follows. The `InitBlock` member `FragmentLocator` is itself a data structure. Going deeper still, you'll find that the `u` field of the `FragmentLocator` is a union type that consists of three structs. One of those structs is named `onDisk`. One of the fields of `onDisk` is `fileSpec`, which is of type `FSSpecPtr`. Now, if you followed all that, take a break and congratulate yourself! If you didn't—and you're still curious (or extremely aggravated)—you'll need to look at the complete definitions of all of these structures in a copy of *Inside Macintosh: PowerPC System Software*.

You can attempt to memorize all of the above `InitBlock` structure information, or, you can simply make note of the fact that this one line of code will always provide you with an `FSSpec` pointer that can be used to access library resources:

```
the_FSSpec_ptr = init_block_ptr->fragLocator.u.onDisk.fileSpec;
```

The remainder of the initialization routine is pretty straightforward stuff. A call to `FSpOpenResFile()` is made in order to get a reference number to the import library's resource fork. This reference number is then used

Programming the PowerPC

in a call to `UseResFile()` to make the import library resource fork the current resource fork. The Toolbox function `Alert()` then posts the alert. Once dismissed, a call to `CloseResFile()` will close the import's resource fork.

If you're tempted to save a little typing, you might try to write the initialization routine as follows:

```
OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )
{
    Alert( 128, nil );

    return ( noErr );
}
```

What would happen if I tried to take the shortcut of simply calling `Alert()` without first opening the import library's resource fork? The Toolbox will search whatever resource file or files are currently open for an 'ALRT' resource with an ID of 128. It may just find one in the application's resource fork—but that isn't the one you want. The moral of this story? Don't try to save keystrokes. And make use of the `InitBlock` data structure that the system has so gratuitously filled and passed to your routine.

Import Library Advantages

What makes an import library that is as simple as the one presented here have the potential to be of practical use? If the alert contains standard information that should appear in a whole suite of programs that you produce, you'll have a simple means of including that alert in each of those programs. Assume that each program in your package of four related applications has a menu item that opens an alert that displays information about your company—information such as the name, address, and telephone number of your company. If the resources and the code for this alert are housed in an import library, then a single copy of that import library can be used by all programs in the suite. If your company ever changes its address or telephone number, the 'DITL' resource in the

import library can be updated and the import library can be recompiled. As part of an upgrade, this single import library could then be distributed to everyone who has one or more of the your programs.

How does the above scenario prove advantageous over the more traditional method of not using a library file, but instead simply including resources in the application's resource fork? In that pre-PowerPC approach, if the information in the alert needs updating, the application source code will need to be recompiled to create a new version of the application. And, if your package consists of several individual programs, each one of them will have to be recompiled so that each one contains the new information. Not only that, but in order to provide each user with the update, you'll have to verify how many different programs in the suite of programs each user has. You'll then have to then send each user a copy of each application. This could easily involve sending out several disks to each user. Using the library method, you'd only have to verify that a person has any one of your programs. Then you'd simply send out a single library file.

As the previous example shows, the true advantage of using import libraries comes in properly choosing what code should become an import library. You should choose code that will be used by more than one of your applications, or code that may need unavoidable periodic changes.

LOADING AND EXECUTING IMPORT LIBRARY CODE

When you consider that an import library can consist of nothing more than an initialization routine, you can see that creating an import library can be quite simple to do. After writing the code, all you need is a compiler that can mark the code as a library fragment rather than an application fragment. The Metrowerks CodeWarrior compiler is such an environment. Before seeing how CodeWarrior performs this feat, you'll need to see an example of how an application loads and executes an import library.

Creating an FSSpec For an Import Library

To load an import library to memory from disk, you'll make a call to the Code Fragment Manager function `GetDiskFragment()`. This routine requires that the directory path to the import library fragment be in the form of a file system specification—an `FSSpec`. So before covering the `GetDiskFragment()` function in detail, I'll get the `FSSpec` creation out of the way. A call to `FSMakeFSSpec()` will take care of that. Assuming I've named my import library `MyImportLib`, the `FSSpec` code will look similar to this snippet:

```
FSSpec  the_FSSpec;  
  
FSMakeFSSpec( 0, 0L, "\pMyImportLib", &the_FSSpec );
```

The first parameter to the Toolbox function `FSMakeFSSpec()` is the volume reference number. A value of 0 indicates the file in question resides on the default, or startup, drive.

The second parameter is the parent directory of the file. The parent directory contains the file in question. If the first parameter has a value of 0, and the file you're concerned with is in the same directory as the application that will use it, you can also use a value of 0 for the ID. Since this second parameter should be a type `long`, you might want to append an `L` to the 0 to force it to occupy the space of a `long`.

The third parameter to `FSMakeFSSpec()` is the name of the file. This parameter can be either a string, as shown above, or a `Str255` variable.

The last parameter is a pointer to an `FSSpec` variable. `FSMakeFSSpec()` will use the information in the first three parameters to create the `FSSpec` and return it in the fourth parameter.

Loading a Library

To load the import library fragment from its disk file to memory, you'll make use of a Code Fragment Manager routine called

GetDiskFragment(). Here's how the universal files header FragLoad.h declares this routine:

```
OSErr GetDiskFragment( FSSpecPtr      fileSpec,
                       long            offset,
                       long            length,
                       Str63           fragName,
                       LoadFlags      findFlags,
                       ConnectionID    *connID,
                       Ptr              *mainAddr,
                       Str255          errName );
```

**TABLE 11.1 THE PARAMETERS TO THE CODE FRAGMENT
MANAGER ROUTINE GetDiskFragment().**

Parameter	Description and typical value
fileSpec	Pointer to the file system specification for the fragment to load.
offset	Offset, in bytes, from the start of the fragment's data fork to the start of the fragment code in the data fork. This will typically be 0.
length	Length, in bytes, of the fragment. A value of 0 specifies that the entire fragment should be loaded—so typically this will be 0.
fragName	The fragment name. This is an optional name used for debugging. Typically this is set to the name of the fragment file.
findFlags	Specifies which of three operations to perform on the fragment. Typically this will be set to the constant kLoadNewCopy.
connID	An ID that identifies the connection to the fragment. Will be used when the application unloads the fragment at a later time. This will be filled in by the function.
mainAddr	The address of the fragment's main special routine. This will be filled in by the function. If the fragment has no main defined, nil will be returned.
errName	If GetDiskFragment() fails, this will be filled in with the name of the fragment that couldn't be loaded.

Programming the PowerPC

At first glance the parameter list for `GetDiskFragment()` looks a little imposing. The truth of the matter, however, is that things aren't nearly as bad as they appear. Once you include a call to `GetDiskFragment()` in your source code, you'll generally be able to copy it and use it "as is" in just about any other program you write. Table 11.1 provides a run down on the purpose of each parameter, as well as a typical value for many of them.

Table 11.1 shows that two of the eight parameters (`offset` and `length`) usually have a value of 0. One of the parameters (`findFlags`) is set to some constant value—usually `kLoadNewCopy`. Two of the parameters get their values from the `FSSpec` structure that holds information about the file system specification for the code fragment file. The remaining three parameters are filled in by `GetDiskFragment()` and returned to the calling routine.

Before calling `GetDiskFragment()`, you'll need to declare a few variables:

```
FSSpec      the_FSSpec;
ConnectionID Lib_Connect_ID = 0;
Ptr         Lib_Main_Ptr   = nil;
OSErr       error          = noErr;
```

The first variable will hold a file system specification—the directory path to the import library fragment. The last three of the variables will hold values returned by `GetDiskFragment()`. Based on Table 11.1, here's what a typical call to `GetDiskFragment()` should look like:

```
error = GetDiskFragment( &the_FSSpec,
                        0,
                        0,
                        the_FSSpec.name,
                        kLoadNewCopy,
                        &Lib_Connect_ID,
                        (Ptr *)&Lib_Main_Ptr,
                        error_name );
```

Assuming I've created an import library named `CompanyInfo`, and it will reside in the same folder as the application fragment that uses it, here's how I could load it into memory:

```
FSSpec      the_FSSpec;
ConnectionID Lib_Connect_ID = 0;
Ptr          Lib_Main_Ptr   = nil;
OSErr        error          = noErr;

FSMakeFSSpec( 0, 0L, "\pCompanyInfo", &the_FSSpec );

error = GetDiskFragment( &the_FSSpec,
                        0,
                        0,
                        the_FSSpec.name,
                        kLoadNewCopy,
                        &Lib_Connect_ID,
                        (Ptr *)&Lib_Main_Ptr,
                        error_name );
```

When the call to `GetDiskFragment()` loads the import library, the import library's initialization routine—if it has one defined—will execute.

Unloading a Library

After `GetDiskFragment()` successfully loads a library, your program will have a connection to that fragment in the form of a variable of type `ConnectionID`. The primary purpose for this variable is to give you a reference to the fragment when you want to unload it from memory. To unload a fragment, pass a pointer to the `ConnectionID` variable to the Code Fragment Manager routine `CloseConnection()`:

```
CloseConnection( &Lib_Connect_ID );
```

This function will call the fragment's termination routine (if it has one defined) and then release the memory allocated to the fragment's code.

CREATING A LIBRARY WITH CODEWARRIOR

If you own the Metrowerks PowerPC compiler and you want to develop shared libraries, you're in luck. The Metrowerks C/C++ PPC compiler

Programming the PowerPC

makes turning source code into a library an easy process—as you’ll see in this section.

Earlier in this chapter you saw an example of an import library that displays an alert. In this section I’ll use CodeWarrior to turn that example into a functional import library.



As of this book’s printing, the Symantec environment does not support the development of shared libraries. If you own the Symantec Cross Development Kit, you’ll want to check with Symantec support—Symantec will soon be releasing a new PowerPC compiler that will support shared libraries.

The Import Library Resources

Import libraries that use resources will want to have those resources self-contained. To make an import library using CodeWarrior you create a resource file, a project file, and a source code file—just as you would for an application. Since my import library will be named `CompanyInfo`, I’ve created a resource file named `CompanyInfo.p.rsrc`. To keep the files I’ll be creating together, I’ve also made a new folder named (11) `TestApp1 Lib f`—you’ll find it on the included disk.

Though an import library isn’t required to include any resources, it should minimally include a ‘vers’ resource. The ‘vers’ resource provides version information about the library. The ‘vers’ resource has an edit box that allows you to add version and copyright information for the application. While that’s the typical use for this text, I’ll instead write a descriptive sentence that will help indicate what this file will be used for. The ‘vers’ resource is shown in Figure 11.3.

After building the library I’ll be able to select **Get Info** from the File menu in the finder to see the string that’s in the ‘vers’ resource. Figure 11.4 shows the string in the Version field of the Get Info window.

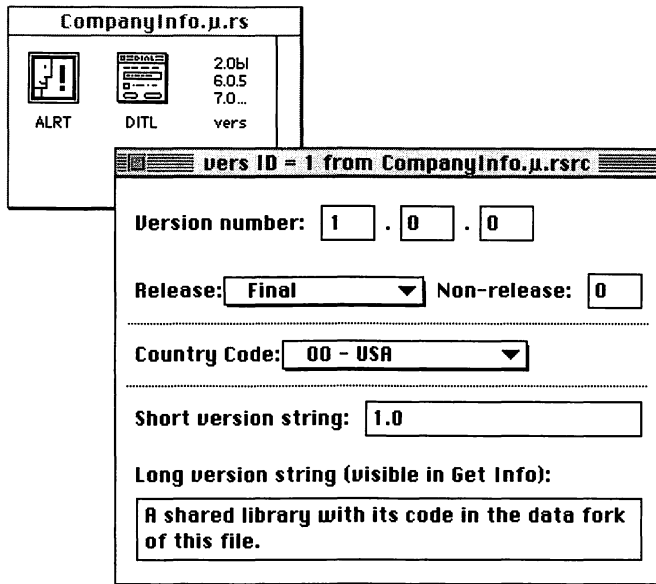


FIGURE 11.3 THE 'VERS' RESOURCE FOR THE COMPANYINFO IMPORT LIBRARY.

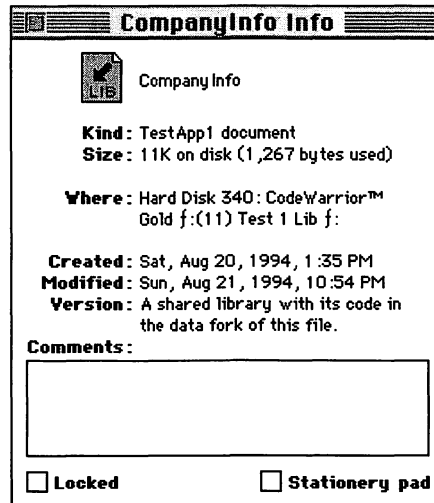


FIGURE 11.4 THE GET INFO DIALOG BOX FOR THE COMPANYINFO IMPORT LIBRARY.

Programming the PowerPC

The import library displays an alert, so I'll need to create an 'ALRT' resource and a 'DITL' resource. Figure 11.5 shows what the 'DITL' for my example looks like.

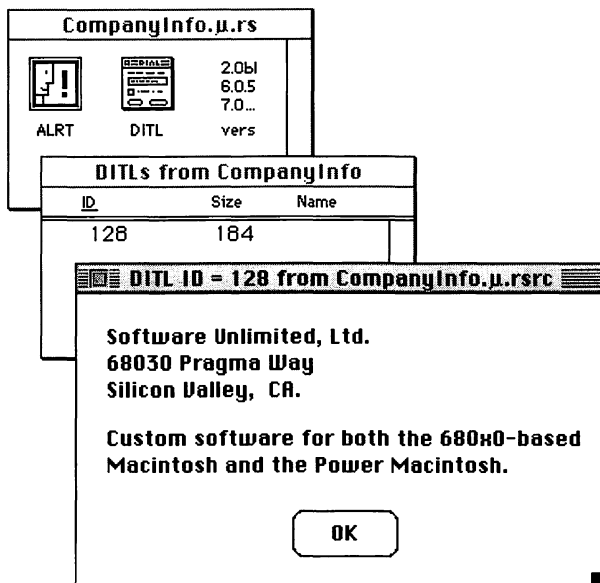
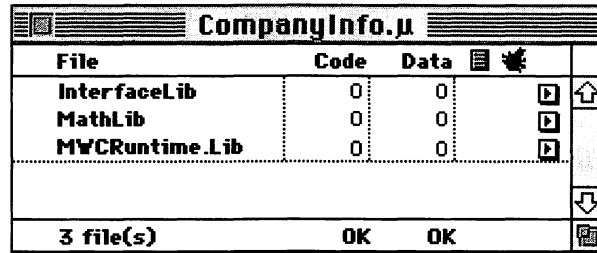


FIGURE 11.5 THE 'DITL' RESOURCE FOR THE COMPANYINFO IMPORT LIBRARY.

With the resources added, I'll save the file and quit the resource editor. Next, I'll need to create a CodeWarrior project to hold the source code for the import library.

The Import Library Project

An import library starts out as a project—just as an application does. I've created a Metrowerks C/C++ PPC project named CompanyInfo.μ—you can see the project window in Figure 11.6. Depending on the code that makes up the import library, you may or may not need some of the three libraries that you add to CodeWarrior application projects—InterfaceLib, MathLib, and MWCRuntime.Lib. In Figure 11.6 you can see that I've gone ahead and added all three of these libraries to the project.



File	Code	Data	
InterfaceLib	0	0	[icon]
MathLib	0	0	[icon]
MYCRuntime.Lib	0	0	[icon]
3 file(s)		OK	OK

FIGURE 11.6 THE COMPANYINFO PROJECT WINDOW.

When you create a new project using CodeWarrior, the compiler assumes that an application will be built from the project. To indicate that this project will instead be a shared library, select **Preferences** from the Edit menu. In the preferences dialog box, click on the **Project** icon to display the Project panel. Use the pop-up menu to change the project from an application to a shared library. That's shown in Figure 11.7.

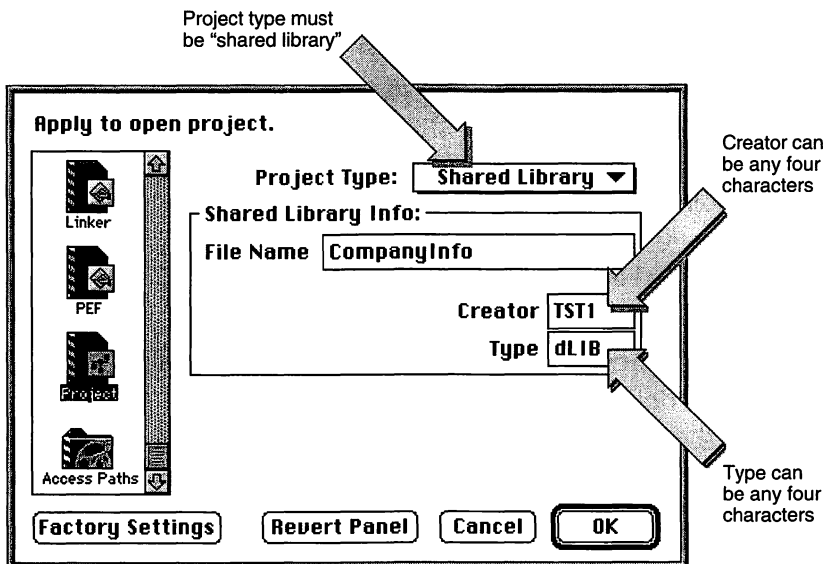


FIGURE 11.7 SETTING A PROJECT'S TYPE TO SHARED LIBRARY USING THE CODEWARRIOR PREFERENCES DIALOG BOX.

Programming the PowerPC

This figure also shows that the file creator and the file type have been changed. Every file has a four character creator. I've selected 'TST1' for "test 1." While the choice of creator is arbitrary, you should set this field to 'TST1' if you want your project to match the remaining figures in this chapter. For an application, the file type must be set to 'APPL.' For a library, the file's type is arbitrary. I've named mine 'dLIB,' hinting that the file will be a library held in the data fork of a file.



Note that the choice of 'dLIB' isn't any kind of Apple or Metrowerks naming convention—it was strictly my choice. If I was creating an application, I would have to use 'APPL' as the file type. If I was creating a text file, I would have to use 'TEXT' as the file type. Since there is no pre-existing CompanyInfo type, I can use whatever characters I want.

There is one combination of characters that will make your import library fall into a specific category of import libraries. If you set the project type to 'shlb', your import library will get a standard shared library icon. You'll see an example of a 'shlb' later in this chapter.

Before dismissing the preferences dialog box, click on the **Linker** icon. That will display the Linker panel, which is shown in Figure 11.8. This panel allows you to specify the three optional entry points into your import library. In Figure 11.8 you can see that these three entry points default to `__initialize`, `__start`, and `__terminate`.

Each entry point corresponds to one of the three special routines that a fragment can define. Entering the names of an import library's special routines lets the linker generate exported symbols for this library. As discussed earlier in this chapter, a PowerPC linker will generate export symbols for a library fragment. When an application fragment is created, the linker will also generate imported symbols in that fragment. The imported symbols it generates will be for the library routines that are called by the application. During run time, it will be the job of the Code Fragment Manager to see to it that the export symbols of shared library routines are paired with import symbols in the application that calls these routines.

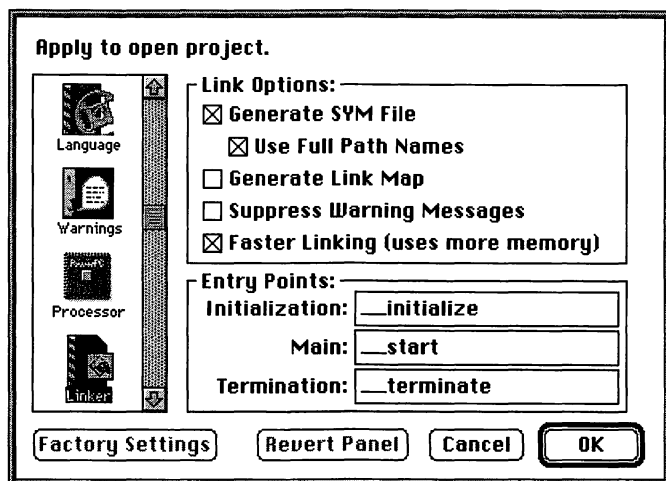


FIGURE 11.8 THE ENTRY POINTS FOR AN IMPORT LIBRARY ARE LISTED IN THE PREFERENCES DIALOG BOX OF CODEWARRIOR.

The Linker panel requests that the entry points be given by function name. So in order to specify the library entry points, I'll need to be familiar with the source code for the import library. Since I've already written the code for the CompanyInfo library, I know that it defines one of the three special routines—an initialization routine named `My_Initialize_Routine()`. I'll enter that name in the Linker panel, and, since my CompanyInfo source code doesn't define either a main routine or a termination routine, I'll make sure that I blank out the other two edit boxes. Figure 11.9 shows the Linker panel after I've listed the initialization routine.

After listing the special routines, the preferences dialog box can be dismissed. The only thing left to do to the project is to add a source code file.

The Import Library Source Code

An import library project needs a source code file, just as an application project does. While still in CodeWarrior, I'll select **New** from the File menu, and then **Save** from the File menu. When prompted for a file

Programming the PowerPC

name, I entered the name `CompanyInfo.c`. After dismissing the Save dialog box, I selected **Add Window** from the Project menu. That added the new source code file to the project window, which now looks like the one shown in Figure 11.10.

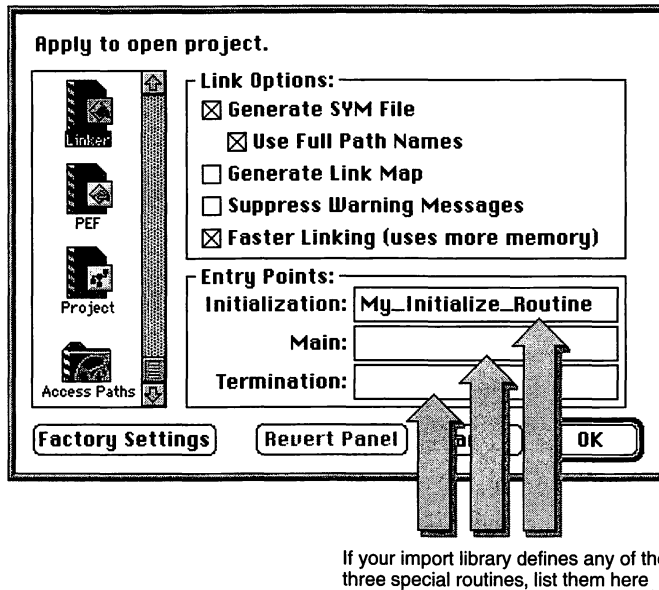


FIGURE 11.9 SETTING THE NAME OF AN IMPORT LIBRARIES INITIALIZATION ROUTINE IN THE CODEWARRIOR PREFERENCES DIALOG BOX.

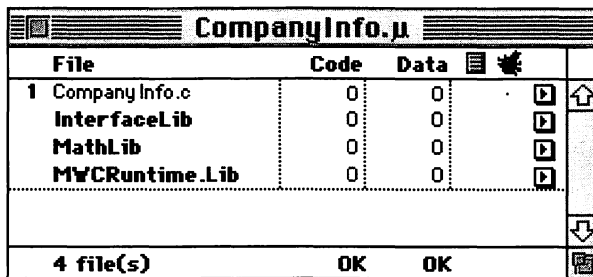


FIGURE 11.10 THE COMPANYINFO PROJECT WINDOW.

Next, I'll type in the library source code—it's listed below. Since I walked through the source code for the CompanyInfo library earlier in this chapter, I'll forego any explanation here. If you have any questions about it, flip back to the section titled Import Library Code.

```
//+++++ include directives ++++++

#include <FragLoad.h>

//+++++ define directives ++++++

#define ABOUT_COMPANY_ALERT 128

//+++++ initialization routine ++++++

OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )
{
    FSSpecPtr the_FSSpec_ptr;
    short res_ref_num;

    the_FSSpec_ptr = init_block_ptr->fragLocator.u.onDisk.fileSpec;
    res_ref_num = FSpOpenResFile( the_FSSpec_ptr , fsCurPerm );
    UseResFile( res_ref_num );

    Alert( ABOUT_COMPANY_ALERT, nil );

    CloseResFile( res_ref_num );

    return ( noErr );
}
```

To build the library, select **Make** from the Project menu—just as you would do if you were building an application from a project. Because you used the preferences dialog box to set the project type to shared library rather than application, the compiler will generate an import library rather than an application. After building the CompanyInfo import library, the folder that holds the project looks like the one pictured in Figure 11.11.

Programming the PowerPC

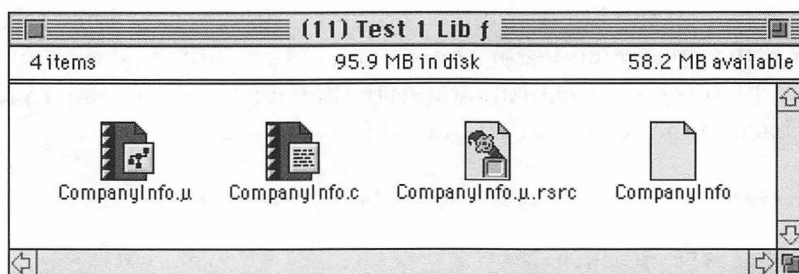


FIGURE 11.11 THE ICON FOR THE COMPANYINFO IMPORT LIBRARY.

In Figure 11.11 you can see that the Finder gives the import library a generic document icon. You and I know, however, that the import library consists of much more than what a typical document normally holds. To take a look, I opened the CompanyInfo library with ResEdit. Note that I opened the library itself—not the Company.u.rsrc resource file that was used for the project. Figure 11.12 shows that the resource fork of the import library consists of the ‘ALRT,’ ‘DITL,’ and ‘vers’ resources that originated in the Company.u.rsrc resource file. Because the import library is a PowerPC file, CodeWarrior added a ‘cfrg’ resource.

To get an idea of what the entire library looks like, I selected **Get Info for CompanyInfo** from the File menu of ResEdit. The dialog box that opened showed that the library’s creator was ‘TST1’ and its type was ‘dLIB’—as set in the CodeWarrior project. More importantly, the dialog box shows that the data fork for the import library isn’t empty—there’s 529 bytes in it. Those bytes are the import library code that posts the alert.

CREATING A TEST APPLICATION WITH CODEWARRIOR

By definition, an import library is not standalone code. So in this section I’ll use CodeWarrior to create a simple application fragment that includes a call to `GetDiskFragment()`. The fragment that `GetDiskFragment()` loads and executes will of course be the CompanyInfo shared library that was developed in the previous section.

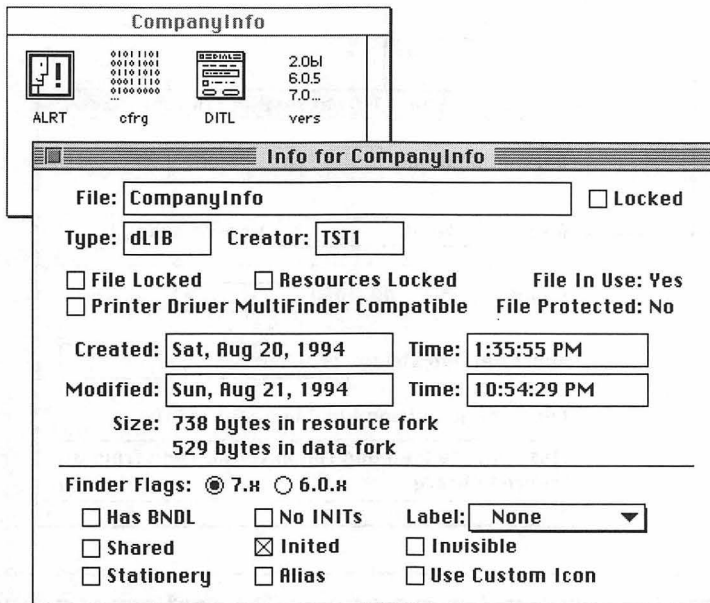


FIGURE 11.12 THE RESOURCE FORK OF THE COMPANYINFO IMPORT LIBRARY, AND THE RESEDIT GET INFO DIALOG BOX FOR THAT LIBRARY.

The Application Resources

The simple test application, which I'll name TestApp1, has no menus, windows, or dialogs—so it requires no resources. However, since System 7 applications typically have a 'vers' resource that gives information about an application, I'll create a resource file that holds a 'vers' resource—it's shown in Figure 11.13.



NOTE

System 7 applications should have a 'SIZE' resource, too. Since the Metrowerks compiler adds this resource during the build of an application, I won't bother adding one myself. Additionally, a PowerPC application should contain a 'cfrg' resource. Again, the Metrowerks compiler will add that to the application when it gets built.

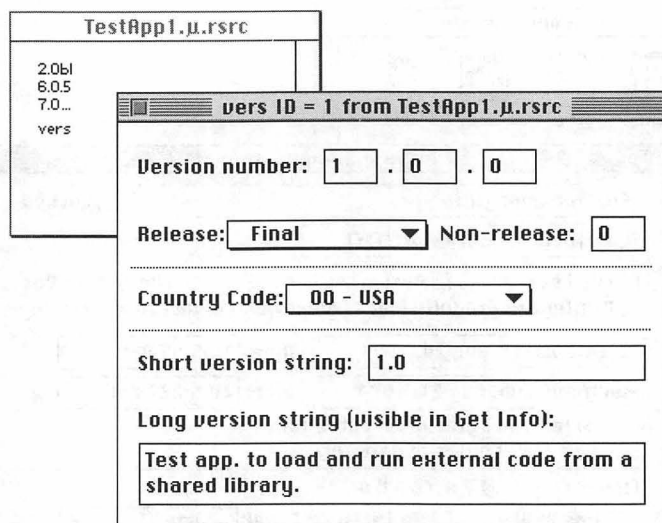


FIGURE 11.13 THE 'VERS' RESOURCE FOR THE TESTAPP1 TEST APPLICATION.

The Application Project

The TestApp1 project contains a single source code file and the three libraries that I add to each of my Metrowerks PowerPC projects. For now, I've created a new, empty source code file named TestApp1.c and added it to the project—I'll describe the source code just ahead a bit. The project window is shown in Figure 11.14.

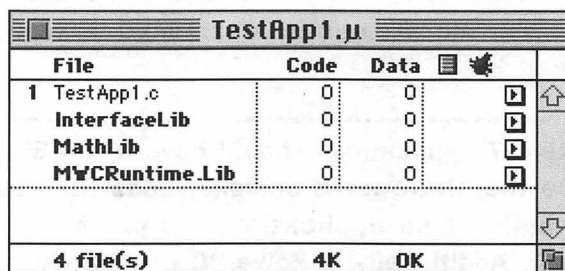


FIGURE 11.14 THE TESTAPP1 PROJECT WINDOW.

Next, I selected **Preferences** from the Edit menu to open the preferences dialog box. I clicked on the **Project** icon to display the Project panel. As it did for a shared library, this panel allows me to set the project type and set the fragment's creator and type. The project type should of course be "Application." The type of any application is 'APPL,' which CodeWarrior has defaulted to. While I can give the fragment any four character creator name I want, I selectively chose to give it the same creator as the import library—"TST1."

The power of an import library is that it can be used by any program that knows how to load it. So an import library doesn't have to have a Creator type that is the same as the application that will be using it. In this example, though, I have given both the library and the test application that will load it the same creator type. I've done this for only one reason—it will make giving the library its own distinctive icon easier. Providing a nongeneric icon for the library won't be covered until Chapter 12—but it can't hurt to plan ahead. Figure 11.15 shows the Project panel for the TestApp1 project.

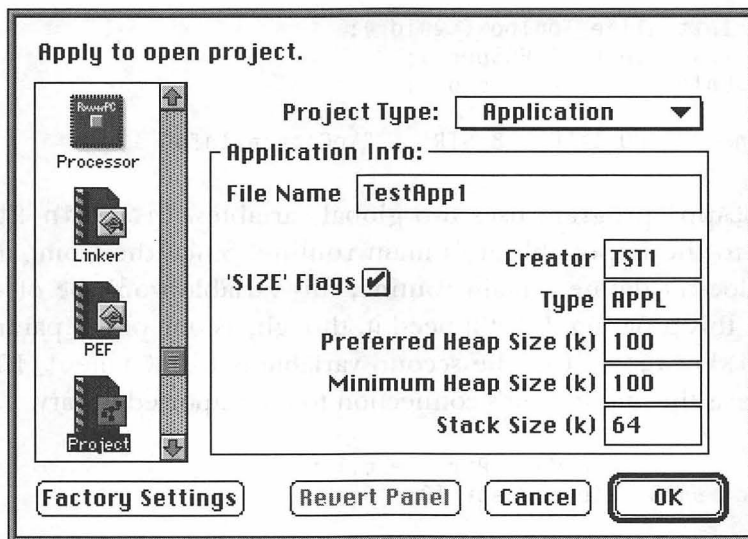


FIGURE 11.15 THE PROJECT PANEL FOR THE TESTAPP1 PROJECT.

The Application Source Code

The TestApp1 application has only one purpose—to see if GetDiskFragment() really does load an import library and automatically execute that library's initialization routine. The TestApp1 application, when launched, will perform the standard Toolbox initializations, then load the CompanyInfo library created in the previous section.

Because the application fragment uses fragment loading code, I'll want to include the FragLoad.h universal header file—just as I did for the import library:

```
#include <FragLoad.h>
```

Next, I'll list the function prototypes for the three application-defined functions. I'll also add a #define directive to the top of my code so that if I ever decide to rename the CompanyInfo import library, I won't have to search through the source code to find any references to it.

```
void Initialize_Toolbox( void );
void Load_Library( FSSpec );
void Unload_Library( void );

#define CO_INFO_LIB_STR    "\\pCompanyInfo"
```

The TestApp1 program uses two global variables. Lib_Main_Ptr is a pointer to the import library's main routine. Since the CompanyInfo library doesn't define a main routine, this variable won't be of significance in this program. I'll still need it, though, as one of the parameters to GetDiskFragment(). The second variable is Lib_Connect_ID. This will serve as the application's connection to the imported library.

```
Ptr          Lib_Main_Ptr    = nil;
ConnectionID Lib_Connect_ID = 0;
```

Next, it's on to some real code. The main() function first calls a routine that performs the standard Toolbox initialization. Then, a call to the Toolbox function FSMakeFSSpec() is made. You'll recall that an import

library is loaded by way of a call to `GetDiskFragment()`, and this function requires a pointer to an `FSSpec`. I'll use the call to `FSMakeFSSpec()` to request that the Toolbox create an `FSSpec` for the import library file. With the `FSSpec` established, I'll call an application-defined function to take care of the actual loading of the import library. Loading the library will kick off the library's initialization routine—which means an alert will be displayed. When the user dismisses the alert, I'll consider my test program finished. I'll call the last application-defined routine, `Unload_Library()`, to unload the import library fragment. Here's a look at the `main()` function:

```
void main( void )
{
    FSSpec  the_FSSpec;
    Initialize_Toolbox();
    FSMakeFSSpec( 0, 0L, CO_INFO_LIB_STR, &the_FSSpec );
    Load_Library( the_FSSpec );
    Unload_Library();
}
```

The `Load_Library()` function is centered around a call to `GetDiskFragment()`. But before calling this routine, I'll make a call to the last of the application-defined routines—`Unload_Library()`. `Unload_Library()` checks to see if there is already an open import library. If there is, it gets unloaded.

The call to `GetDiskFragment()` should look familiar. It's eight parameters were all discussed in this chapter's Loading and Executing Import Library Code section. When the call is complete, the error variable should have a value of `noError`. If it doesn't, `Load_Library()` unloads the problem library and exits.



You'll want to consider a more graceful way of handling a failed library load. After unloading the library, you could post an alert that gave the user an informative message based on the value of the error variable.

Programming the PowerPC

```
void Load_Library( FSSpec the_FSSpec )
{
    OSErr    error = noErr;
    Str255    error_name;

    Unload_Library();

    error = GetDiskFragment( &the_FSSpec,
                             0,
                             0,
                             the_FSSpec.name,
                             kLoadNewCopy,
                             &Lib_Connect_ID,
                             (Ptr *)&Lib_Main_Ptr,
                             error_name );

    if ( error != noErr )
    {
        Unload_Library();
        ExitToShell();
    }
}
```

The `Unload_Library()` function first checks to see if there is a valid connection to a library. If `Lib_Connect_ID` has a nonzero value, then there is. The Code Fragment Manager routine `CloseConnection()` is then called to unload the fragment. Before the routine ends, the global variable `Lib_Connect_ID` is set to 0 to show that no library is open.

```
void Unload_Library()
{
    if ( Lib_Connect_ID != 0 )
    {
        CloseConnection( &Lib_Connect_ID );
        Lib_Connect_ID = 0;
    }
}
```

Here's an uninterrupted look at the source code for `TestApp1`. Once you grasp how this application works, you're well on your way to understanding PowerPC import libraries.

```
//+++++ include directives ++++++
#include <FragLoad.h>

//+++++ function prototypes ++++++
void Initialize_Toolbox( void );
void Load_Library( FSSpec );
void Unload_Library( void );

//+++++ define directives ++++++
#define CO_INFO_LIB_STR "\pCompanyInfo"

//+++++ global variables ++++++
Ptr      Lib_Main_Ptr  = nil;
ConnectionID Lib_Connect_ID = 0;

//+++++ main ++++++
void main( void )
{
    FSSpec the_FSSpec;

    Initialize_Toolbox();

    FSMakeFSSpec( 0, 0L, CO_INFO_LIB_STR, &the_FSSpec );

    Load_Library( the_FSSpec );

    Unload_Library();
}

//+++++ initialize the Toolbox ++++++
void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
}
```

Programming the PowerPC

```
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();
}

//+++++++ load a library ++++++++

void Load_Library( FSSpec the_FSSpec )
{
    OSErr    error = noErr;
    Str255   error_name;

    Unload_Library();

    error = GetDiskFragment( &the_FSSpec,
                             0,
                             0,
                             the_FSSpec.name,
                             kLoadNewCopy,
                             &Lib_Connect_ID,
                             (Ptr *)&Lib_Main_Ptr,
                             error_name );

    if ( error != noErr )
    {
        Unload_Library();
        ExitToShell();
    }
}

//+++++++ unload a library ++++++++

void Unload_Library()
{
    CloseConnection( &Lib_Connect_ID );
}
```

Once I've entered all the source code, I'll select **Make** from the Project menu. The result will be a PowerPC application named TestApp1. Figure 11.16 shows the folder that houses both the TestApp1 project and the CompanyInfo project.

If you open the TestApp1 application (not the TestApp1.p.rsrc file) with ResEdit you'd see that CodeWarrior added the required 'cfrg' resource—as well as a 'SIZE' resource—to the 'vers' resource that came

from the `TestApp1.u.rsrc` file. That's shown in Figure 11.17. Note that there are no 'CODE' resources in the application. I'm only looking at the resource fork of the application, while the code is in the data fork.

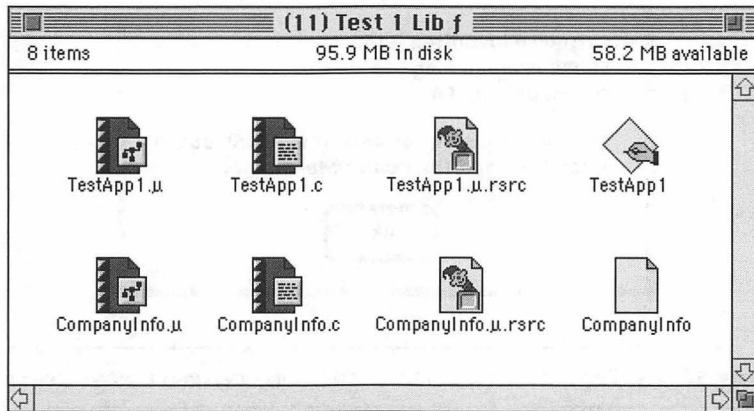


FIGURE 11.16 THE FOLDER THAT HOLDS BOTH THE TEST APPLICATION AND THE IMPORT LIBRARY.

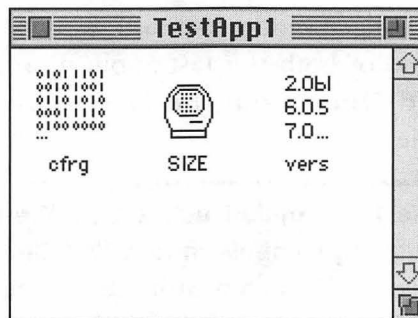


FIGURE 11.17 THE RESOURCE FORK OF THE TESTAPP1 TEST APPLICATION.

Executing the Application and the Library

Running the TestApp1 code will cause the CompanyInfo library to get loaded, and its initialization routine code to execute. The call to the

Programming the PowerPC

application-defined routine `Load_Library()` takes care of that. When I ran `TestApp1` I saw the alert that's pictured in Figure 11.18.

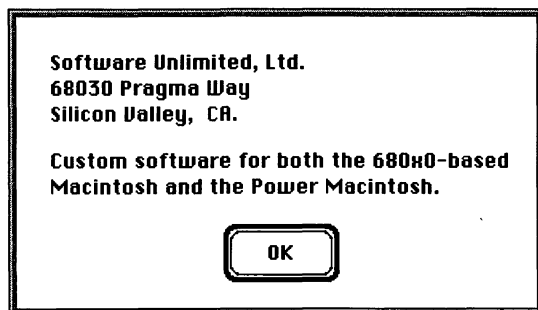


FIGURE 11.18 THE ALERT DISPLAYED BY THE COMPANYINFO IMPORT LIBRARY.

Clicking the **OK** button dismisses the alert and ends the program. To see the import library code execute, I again ran `TestApp1`—but this time from within the CodeWarrior environment. Holding the **option** key down while selecting **Run** from the Project menu starts the debugger. Instead of just one debugger window open, I want two windows—I want to view the source code of both the test application and of the import library. So I selected **Open** from the File menu and opened the `CompanyInfo.xSYM` file.



When a file is compiled using CodeWarrior, the compiler saves debugging symbols in a `.xSYM` file. The file will have the source code file name with `.xSYM` appended to it.

Next, I set two breakpoints. I set the first at the call to `Load_Library()` in `main()` of the test application. The second was set at the first executable line in the initialization routine of the `CompanyInfo` import library. These breakpoints are shown in Figure 11.19.

Selecting **Run** from the Control menu starts the `TestApp1` application running. It stops at the breakpoint by the call to `Load_Library()`. The arrow in Figure 11.20 indicates this.

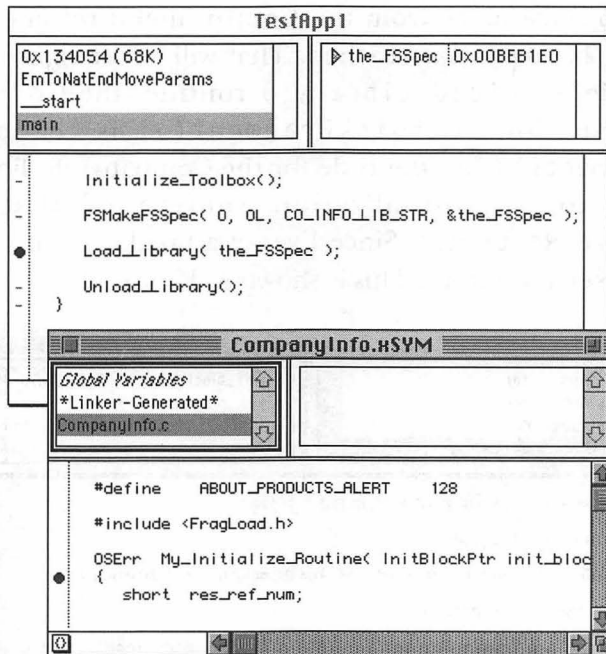


FIGURE 11.19 SETTING BREAKPOINTS IN BOTH THE TEST APPLICATION AND THE IMPORT LIBRARY.

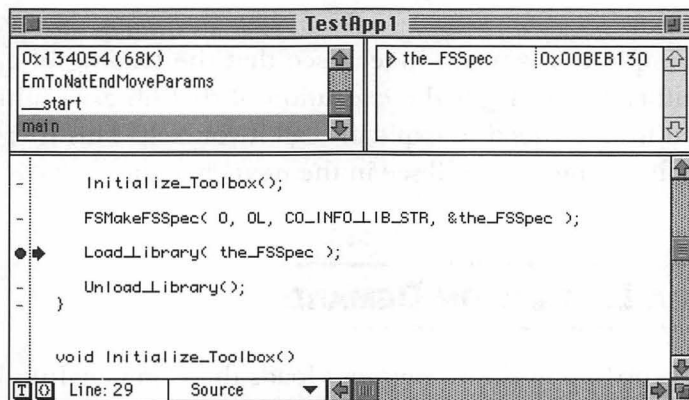


FIGURE 11.20 BREAKING JUST BEFORE THE CALL TO `Load_Library()`.

Programming the PowerPC

Selecting **Run** once more from the Control menu resumes execution until the next breakpoint is reached. That will occur right away. In the application-defined `Load_Library()` routine, the Code Fragment Manager function `GetDiskFragment()` is called. When `GetDiskFragment()` loads the code for the `CompanyInfo` library, execution jumps to the initialization routine of that library—`My_Initialize_Routine()`. Since I've set a breakpoint in this routine, that's where execution stops. This is shown in Figure 11.21.

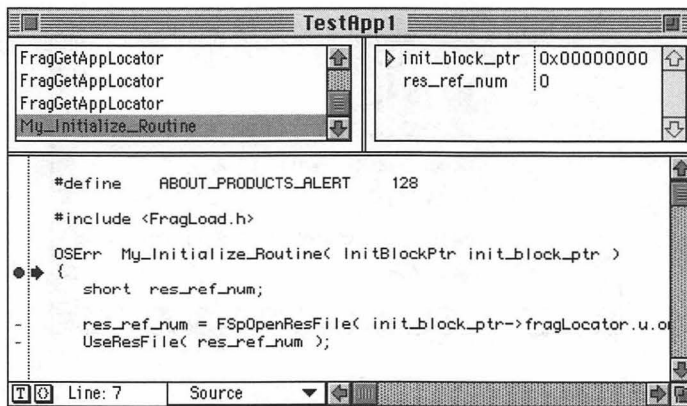


FIGURE 11.21 BREAKING IN THE IMPORT LIBRARY'S INITIALIZATION ROUTINE.

From this simple test you can indeed see that the loading of an import library is sufficient to trigger the execution of that library's initialization routine. You'll never need to explicitly call it yourself. This *isn't* true of a library's main routine, as you'll see in the next chapter.

LOADING A LIBRARY ON DEMAND

The TestApp1 application fragment loads the `CompanyInfo` library at application startup. This doesn't have to be, and quite often isn't, the case. Your applications can make use of library code only when each application requires that code. The most obvious way to implement this

conditional loading is through the use of a menu item. In this section I'll modify TestApp1 to include a menu bar and menus to do just that. I'll name the resulting application TestApp2.

The Test Application's Resources

TestApp2 will have a menu bar with three menus—their 'MENU' resources are shown in Figure 11.22. I'll tie the three 'MENU' resources together using a 'MBAR' resource—shown in Figure 11.23.

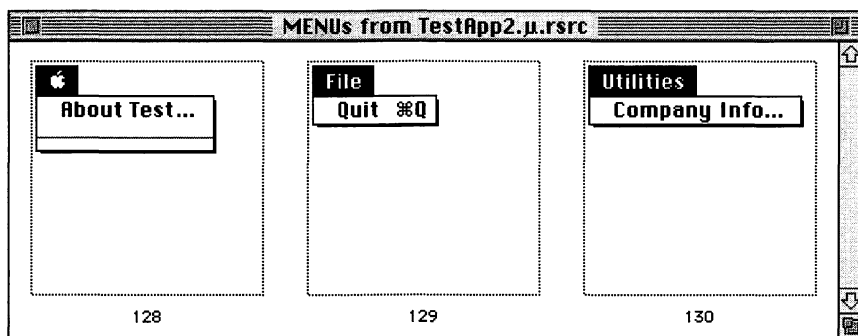


FIGURE 11.22 THE 'MENU' RESOURCES FOR TESTAPP2.

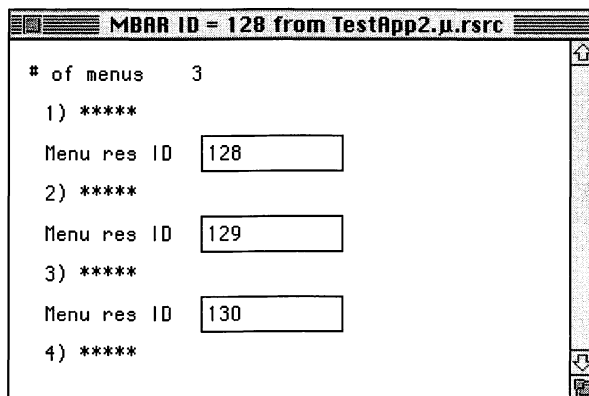


FIGURE 11.23 THE 'MBAR' RESOURCE FOR TESTAPP2.

Programming the PowerPC

The Apple menu will have the standard About item. I've included an 'ALRT' and 'DITL' resource to handle that menu item. The 'DITL' is shown in Figure 11.24.

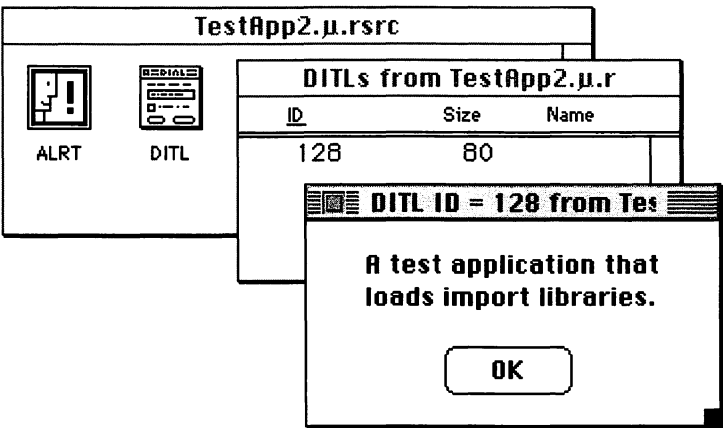


FIGURE 11.24 THE 'DITL' RESOURCE FOR TESTAPP2.

The File menu will be used to quit the program, while the Utilities menu will be used to give the user the opportunity to load the CompanyInfo library. Figure 11.25 shows all the resource types in the TestApp2 resource file.

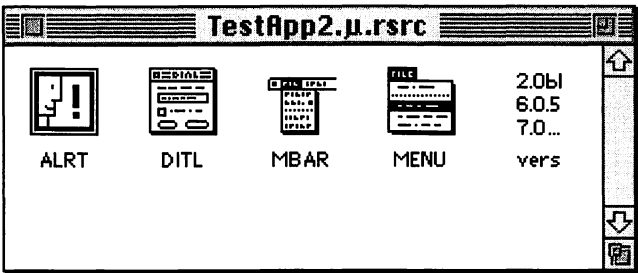


FIGURE 11.25 THE RESOURCE TYPES FOUND IN TESTAPP2.

The Argument for Import Libraries

For simplicity, TestApp2 has just a single menu item that loads an import library. But this idea could easily be extended. Add-on tools, or utilities, are becoming a popular feature of many commercial applications. Figure 11.26 shows the Tools menu of the word processor Microsoft Word.

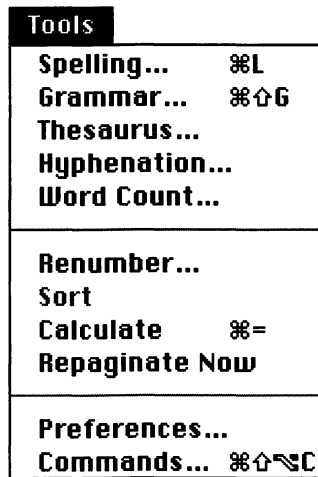
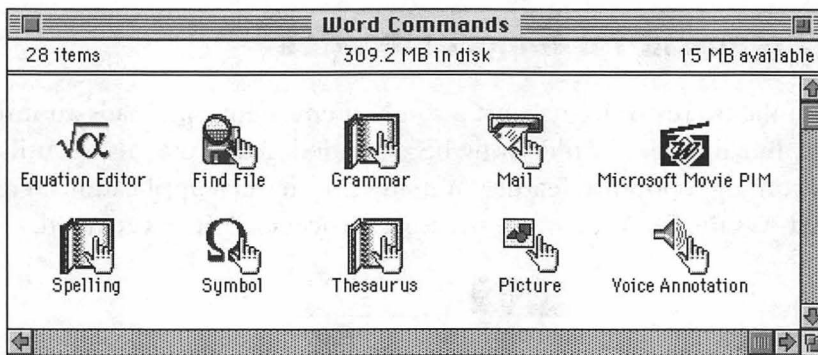


FIGURE 11.26 THE TOOLS MENU FROM THE MICROSOFT WORD WORD PROCESSOR.

If you have Microsoft Word, you'll notice that the installation of that software package added a Word Commands folder to your hard drive. Figure 11.27 shows some of that folder's contents. Note in the figure that some of the icons, such as Spelling and Thesaurus, correspond to menu items from the Word Tools menu.

If Microsoft updates its Word Thesaurus, it only has to make the new Thesaurus document available to the public. The Word application itself need not be redistributed. The new Thesaurus can be sent to registered users, or posted to electronic bulletin boards. Software piracy isn't a concern because without owning the Word application, the Thesaurus itself is useless.



**FIGURE 11.27 THE WORD COMMANDS FOLDER THAT
IS A PART OF THE MICROSOFT WORD PACKAGE.**

A word processor isn't the only type of application that could be enhanced by a utilities-type menu. So TestApp2 serves as a foundation for a program that includes a Tools (or Utilities, or Plug-Ins, or Options, etc.) menu. Here's a couple of ideas:

- A mathematical program might keep different equation-solving algorithms as separate libraries. If an inaccurate algorithm is ever found, it can easily be replaced. Or, if a faster or more precise one is found, it can be substituted for the original one.
- As the typical Mac's memory and processor speed both increase, an existing menu item could be enhanced to take advantage of this power. Consider a program that had a Play Movie menu item that allows the user to select and play a single QuickTime movie. With the assumption that users now have increased memory and processor speed, the import library that held the movie-playing code could be altered to give the user the option of opening two or more movies simultaneously.

The Test Application's Code

TestApp2 differs from TestApp1 in just one respect—it has menus and a menu bar. So I'll keep the source code walk-through to a minimum, with emphasis placed on the differences from TestApp1. I won't skim through the code too quickly, however. As mentioned, TestApp2 serves as a good shell for any program that will make use of libraries "on demand."

TestApp2 begins by including FragLoad.h. Next, a function prototype appears for each function. Then a host of #define directives are added to keep numbers out of the source code listing. Most of the directives are menu resource IDs. The preliminaries end with the declaration of three global variables. You saw Lib_Main_Ptr and Lib_Connect_ID in TestApp1. The third variable is All_Done, used to signal the end of the program.

```
//+++++ include directives ++++++

#include <FragLoad.h>

//+++++ function prototypes ++++++

void Initialize_Toolbox( void );
void Load_Library( FSSpec );
void Unload_Library( void );

void Set_Up_Menu_Bar( void );
void Main_Event_Loop( void );
void Handle_Mouse_Down( EventRecord );
void Handle_Menu_Choice( long );
void Handle_Apple_Choice( short );
void Handle_File_Choice( short );
void Handle_Utility_Choice( short );
FSSpec Get_File_Spec( Str255 );

//+++++ define directives ++++++

#define CO_INFO_LIB_STR "\pCompanyInfo"

#define MENU_BAR_ID 128
```

Programming the PowerPC

```
#define      APPLE_MENU_ID          128
#define      SHOW_ABOUT_ITEM        1
#define      FILE_MENU_ID           129
#define      QUIT_ITEM              1
#define      UTILITY_MENU_ID        130
#define      CO_INFO_ITEM           1

#define      ABOUT_ALERT_ID         128

//+++++++ global variables ++++++

Ptr          Lib_Main_Ptr   = nil;
ConnectionID Lib_Connect_ID = 0;
Boolean      All_Done = false;
```

The program's `main()` routine initializes the Toolbox, sets up the menu bar, then calls `Main_Event_Loop()` to loop repeatedly until the program ends:

```
//+++++++ main ++++++

void main( void )
{
    Initialize_Toolbox();

    Set_Up_Menu_Bar();

    Main_Event_Loop();
}
```

You've seen `Initialize_Toolbox()` in the past, so I'll omit its code here. `Set_Up_Menu_Bar()` uses the standard Menu Manager Toolbox calls to set up the menu bar, get a handle to the Apple menu, add the Apple items to it, and draw the menu bar at the top of the screen:

```
//+++++++ display menu bar ++++++

void Set_Up_Menu_Bar( void )
{
    Handle      menu_bar_handle;
    MenuHandle   apple_menu;

    menu_bar_handle = GetNewMBar( MENU_BAR_ID );
```

```
SetMenuBar( menu_bar_handle );
DisposHandle( menu_bar_handle );

apple_menu = GetMHandle( APPLE_MENU_ID );

AddResMenu( apple_menu, 'DRVr' );

DrawMenuBar();
}
```

`Main_Event_Loop()` contains no surprises. It relies on `WaitNextEvent()` to store information about the most recent event. A `MouseDown` event is sent to `Handle_Mouse_Down()` for further processing. A `KeyDown` event that includes a press of the command key is assumed to be an attempt to access a menu item, and is sent to `Handle_Menu_Choice()`.

```
//+++++ repeat until done +++++
void Main_Event_Loop( void )
{
    EventRecord the_event;
    char        the_key;
    long        menu_choice;

    while ( All_Done == false )
    {
        WaitNextEvent( everyEvent, &the_event, 15L, nil );

        switch ( the_event.what )
        {
            case mouseDown:
                Handle_Mouse_Down( the_event );
                break;

            case keyDown:
                the_key = ( the_event.message & charCodeMask );

                if ( ( the_event.modifiers & cmdKey ) != 0 )
                {
                    menu_choice = MenuKey( the_key );
                    Handle_Menu_Choice( menu_choice );
                }
                break;
        }
    }
}
```

Programming the PowerPC

```
    }  
}
```

TestApp2 is interested in mouse clicks in the menu bar, so that's what `Handle_Mouse_Down()` looks for. When that happens, `Handle_Menu_Choice()` is called to determine which menu was selected.

```
//+++++++ handle a click of the mouse button ++++++  
  
void Handle_Mouse_Down( EventRecord the_event )  
{  
    WindowPtr    the_window;  
    short         the_part;  
    long          menu_choice;  
  
    the_part = FindWindow( the_event.where, &the_window );  
  
    switch ( the_part )  
    {  
        case inMenuBar:  
            menu_choice = MenuSelect( the_event.where );  
            Handle_Menu_Choice( menu_choice );  
            break;  
  
        case inSysWindow:  
            SystemClick( &the_event, the_window );  
            break;  
    }  
}
```

`Handle_Menu_Choice()` is simply a junction point. It determines which menu the mouse click occurred in, and then calls another application-defined routine to actually respond to that menu selection. The Toolbox routine `HiWord()` extracts the number of the selected menu from the `menu_choice` variable, while its companion routine `LoWord()` extracts the number of the selected item in that menu.

```
//+++++++ handle a click in the menu bar ++++++  
  
void Handle_Menu_Choice ( long menu_choice )  
{  
    short the_menu;  
    short the_menu_item;
```

```
if ( menu_choice != 0 )
{
    the_menu = HiWord( menu_choice );
    the_menu_item = LoWord( menu_choice );

    switch ( the_menu )
    {
        case APPLE_MENU_ID:
            Handle_Apple_Choice( the_menu_item );
            break;

        case FILE_MENU_ID:
            Handle_File_Choice( the_menu_item );
            break;

        case UTILITY_MENU_ID:
            Handle_Utility_Choice( the_menu_item );
            break;
    }
    HiliteMenu( 0 );
}
}
```

Handle_Apple_Choice() is standard Apple menu handling code. An About item menu selection posts the alert that I've included in the resource file. Any other menu item selection will be taken care of by the operating system via a call to OpenDeskAcc().

```
//+++++++ handle a click in the Apple menu +++++++

void Handle_Apple_Choice( short the_item )
{
    Str255      desk_acc_name;
    short       desk_acc_number;
    MenuHandle  apple_menu;

    switch ( the_item )
    {
        case SHOW_ABOUT_ITEM:
            Alert( ABOUT_ALERT_ID, nil );
            break;

        default:
            apple_menu = GetMHandle( APPLE_MENU_ID );
    }
}
```

Programming the PowerPC

```
        GetItem( apple_menu, the_item, desk_acc_name );
        desk_acc_number = OpenDeskAcc( desk_acc_name );
        break;
    }
}
```

Handle_File_Choice() is used to quit the application. Setting the All_Done flag to true will cause the main event loop to terminate and end the program.

```
//+++++++ handle a click in the File menu +++++++
void Handle_File_Choice( short the_item )
{
    switch ( the_item )
    {
        case QUIT_ITEM:
            All_Done = true;
            break;
    }
}
```

Handle_Utility_Choice() is used to load the proper import library. While TestApp2 only recognizes one library, this could be easily changed by adding more case labels—and, of course, more menu items to the Utilities 'MENU' resource. In preparation for a future enhancement of this type, Handle_Utility_Choice() calls an application-defined function named Get_File_Spec() to get the FSSpec that Load_Library() needs. The addition of a new case label would require only that the proper file name be passed to Get_File_Spec().

```
//+++++++ handle a click in the Utilities menu +++++
void Handle_Utility_Choice( short the_item )
{
    FSSpec the_FSSpec;

    switch ( the_item )
    {
        case CO_INFO_ITEM:
            the_FSSpec = Get_File_Spec( CO_INFO_LIB_STR );
            Load_Library( the_FSSpec );
            break;
    }
}
```

```
    }  
}
```

Get_File_Spec() accepts a Str255 variable as its only parameter. It uses that string to set up a file standard specification, then returns that FSSpec to the calling routine.

```
//+++++++ return an FSSpec for a file ++++++  
  
FSSpec Get_File_Spec( Str255 the_lib_name )  
{  
    FSSpec the_FSSpec;  
    OSErr error;  
  
    error = FMakeFSSpec( 0, 0L, the_lib_name, &the_FSSpec );  
  
    return ( the_FSSpec );  
}
```

The TestApp2 program concludes with Load_Library() and Unload_Library(). Both of these functions are identical to the versions developed for the TestApp1 program earlier in this chapter.

```
//+++++++ load a library ++++++  
  
void Load_Library( FSSpec the_FSSpec )  
{  
    OSErr error = noErr;  
    Str255 error_name;  
  
    Unload_Library();  
  
    error = GetDiskFragment( &the_FSSpec,  
                             0,  
                             0,  
                             the_FSSpec.name,  
                             kLoadNewCopy,  
                             &Lib_Connect_ID,  
                             (Ptr *)&Lib_Main_Ptr,  
                             error_name );  
  
    if ( error != noErr )  
    {  
        Unload_Library();  
        ExitToShell();  
    }  
}
```

Programming the PowerPC

```
    }  
}  
  
//+++++++ unload a library ++++++++  
  
void Unload_Library()  
{  
    if ( Lib_Connect_ID != 0 )  
    {  
        CloseConnection( &Lib_Connect_ID );  
        Lib_Connect_ID = 0;  
    }  
}
```

SHARING IMPORT LIBRARIES

One of the primary advantages to using import libraries is that you can write one code fragment that can be used—without modification—by two or more application fragments. If you’re going to do that, you won’t want to keep a copy of each import library in a folder with each application. Instead, you should consider making the shared library a shared library (‘shlb’). Then, a single copy of it can be stored in the user’s Extensions folder where it can be accessed by each application.

Sharing the CompanyInfo Library Between Applications

Before creating a ‘shlb,’ I’ll verify that the CompanyInfo library can truly be shared among applications. I put a copy of both the TestApp1 and TestApp2 applications in a new folder—along with a single copy of the CompanyInfo library. This folder is shown in Figure 11.28. Then I ran TestApp1. It opened the alert found in CompanyInfo. Next, I launched TestApp2. I selected **Company Info** from the Utilities menu, and the alert again opened. Success!

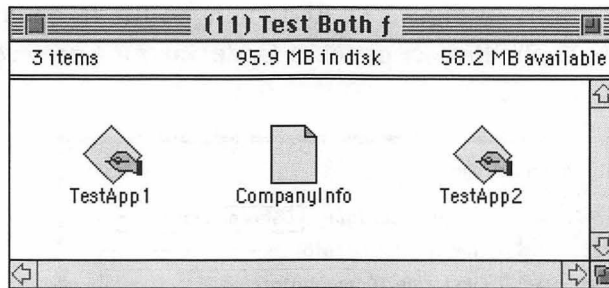


FIGURE 11.28 A FOLDER CONTAINING TWO APPLICATIONS THAT CAN BOTH USE THE CODE IN THE COMPANYINFO IMPORT LIBRARY.

Creating a 'shlb' Library

If you have a Power Mac, and you look in the Extensions folder of your System Folder, you might find a couple of documents with icons like the one pictured in Figure 11.29. This is the standard icon that the Finder gives to a shared library whose type is 'shlb.'

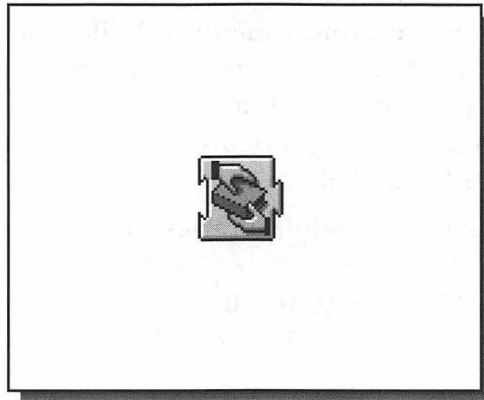


FIGURE 11.29 AN IMPORT LIBRARY OF TYPE 'SHLB' HAS ITS OWN APPLE-DEFINED ICON.

You can make any shared library a 'shlb' library by simply setting its type to 'shlb,' and then recompiling the library. That's what I'm doing to the

Programming the PowerPC

CompanyInfo library in Figure 11.30. I changed the fragment type from 'dLIB' to 'shlb' in the Project panel of CodeWarrior's preferences dialog box.

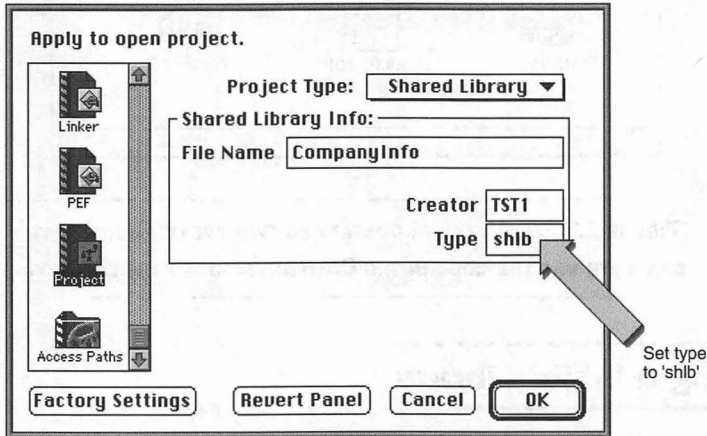


FIGURE 11.30 SETTING A FILE'S TYPE TO 'SHLB.'

I made no changes to the source code for the library. Instead, I just selected **Make** from the Project menu to build a new version of the CompanyInfo library. When complete, the library had a different icon. Figure 11.31 shows a folder with a copies of all of the files for the TestApp1 project and CompanyInfo project. Note that the import library now has the standard shared library icon.

If you're going to use 'shlb' libraries, you'll want to change your application's call to the Code Fragment Manager routine `GetDiskFragment()`. Instead, you'll want to use the Code Fragment Manager function `GetSharedLibrary()`. Here's the prototype for that function:

```
OSErr GetSharedLibrary( Str63      libName,  
                       OSType      archType,  
                       LoadFlags  findFlags,  
                       ConnectionID *connID,  
                       Ptr          *mainAddr,  
                       Str255      errName );
```

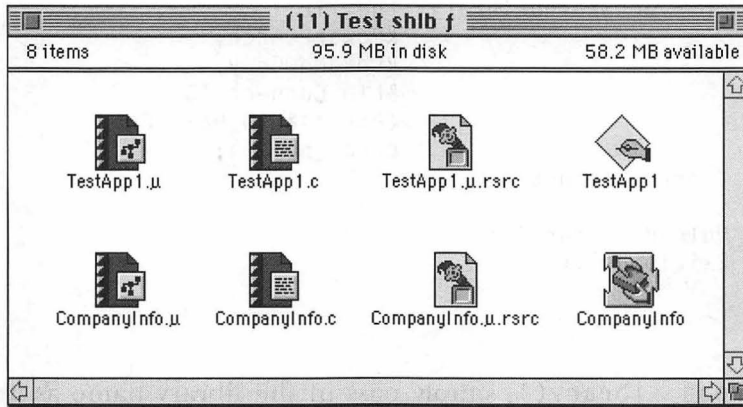


FIGURE 11.31 A FOLDER HOLDING THE NEW TESTAPP1 PROJECT FILES AND THE NEW COMPANYINFO PROJECT FILES.

The first parameter is a string that holds the shared library's name. Though it's defined as a `Str63` type, you can pass in a `Str255` type string if the library name has previously been defined as such. The second parameter specifies the instruction set architecture of the shared library. I've been compiling with a PowerPC compiler, so this parameter should be set to the constant `kPowerPCArch`. The last four parameters to `GetSharedLibrary()` are the same as the last four parameters to `GetDiskFragment()`.

To take advantage of my 'shlb' library, I modified `Load_Library()`. Rather than pass in a `FSSpec`, the routine now accepts the library name as a string. And in place of the call to `GetDiskFragment()` is a call to `GetSharedLibrary()`. Here's the new version of that `TestApp1` and `TestApp2` routine:

```
void Load_Library( Str255 the_lib )
{
    OSErr    error = noErr;
    Str255    error_name;

    Unload_Library();
```

Programming the PowerPC

```
error = GetSharedLibrary( the_lib,
                          kPowerPCArch,
                          kLoadNewCopy,
                          &Lib_Connect_ID,
                          (Ptr *)&Lib_Main_Ptr,
                          error_name );

if ( error != noErr )
{
    Unload_Library();
    ExitToShell();
}
}
```

To call `Load_Library()`, simply pass in the library name as a string. Don't first create an `FSSpec`. Here's a typical call to `Load_Library()`:

```
#define      CO_INFO_LIB_STR      "\\pCompanyInfo"

void main( void )
{
    Initialize_Toolbox();

    Load_Library( CO_INFO_LIB_STR );

    Unload_Library();
}
```

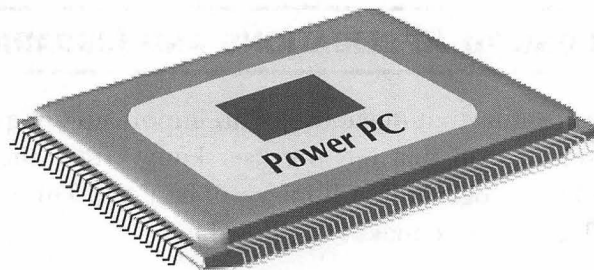
To test out the 'shlb' library type, you can make the above changes to `TestApp1` and to the `CompanyInfo` library. Or you can look in the (11) Test shlb *f* folder—it holds new versions of the project files an application and library. Try running the `TestApp1` program to verify that the new `CompanyInfo` library loads. Then, drag the library to your closed System folder. The system views the library as an extension, and asks if you want to add it to the Extensions folder in the System Folder. Go ahead and add the library. Then restart your Power Mac. Now, thanks to the inclusion of the `GetSharedLibrary()` call in `TestApp1`, that application will be able to find and load the `CompanyInfo` library.

CHAPTER SUMMARY

An import library—also known as a shared library—is a code fragment that relies on some other code fragment to load it. The primary advantage of turning code into an import library rather than an application is that as a shared library the code is accessible by any number of other code fragments. This reduces redundancy in writing code, and makes the chore of updating code an easier task to perform.

You'll use your development environment to mark the outcome of a project to be a shared library rather than an application. Then, you'll include a call to the Code Fragment Manager routine `GetDiskFragment()` in an application fragment. `GetDiskFragment()` loads a specified import library and executes that library's initialization routine—a special import library routine that serves as an entry point into the library.

An import library that has a type of 'shlb' is a special type of shared library. The Finder will give the library a standard shared library icon, and the import library code will be accessible by your applications even when the library is located in a different folder than the application. Typically, 'shlb' libraries are kept in the Extensions folder of your System Folder.



CHAPTER 12

MORE IMPORT LIBRARIES

In Chapter 11 you learned the basics of import libraries. Because the idea of plug-in tools is fast becoming a very popular Macintosh programming concept, this chapter is also devoted to import libraries. The Power Macintosh and the new Code Fragment Manager now give your applications the ability to include this powerful programming feature. And, with the CodeWarrior PowerPC compiler, creating import, or shared, libraries is remarkably simple.

In this chapter you'll see how to give your import libraries their own distinctive icons. You'll also see how to add sophistication to a library by supplying it with three entry points—routines that make the library code accessible by an application fragment. Finally, you'll incorporate Apple events in your application fragment so that an application and an import library can communicate with one another via the Finder.

ADDING ICONS TO APPLICATIONS AND LIBRARIES

Most applications that make use of an import library give that library a distinctive icon that lets the user know either the library's purpose, or by which program that library will be used. This section looks at how the 'BNDL' resource makes this possible.

Adding an Icon to the Application

To create all of the icons I'll need, I only need to add a 'BNDL' resource to one of my applications. As I add and edit icons from the 'BNDL' editor, ResEdit will add the necessary resources to the resource file. I'll begin by selecting **Create New Resource** from the Resource menu. In the Select New Type dialog box that appears, I'll double-click on the 'BNDL' resource. That brings up the BNDL editor, shown in Figure 12.1.

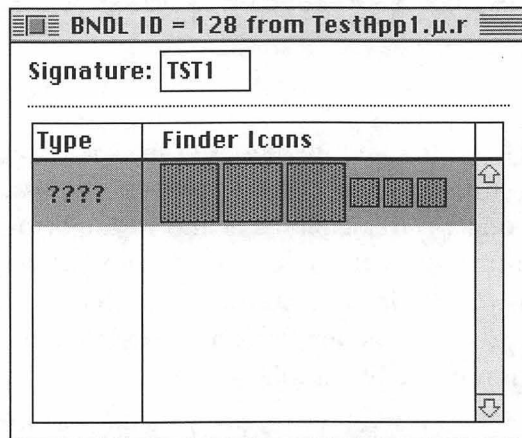


FIGURE 12.1 THE 'BNDL' RESOURCE FOR TESTAPP1.

The signature should be the same as the creator of the application. I'm going to add an icon to my TestApp1 program, so the signature should be 'TST1.' Recall from Chapter 11 that 'TST1' is the creator I used in the

preferences dialog box for the TestApp1 project. Figure 12.1 shows that I've typed the proper signature in the Signature edit box in the editor.

Next, I choose **Create New File Type** from the Resource menu, hit the **tab** key, and type in the four character file type. I'm creating an icon for an application, so this type is 'APPL'. Then I double-click on any of the shaded boxes under the Finder Icons heading—as shown in Figure 12.2.

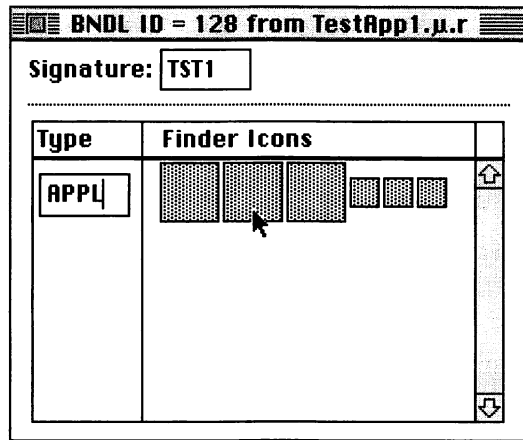


FIGURE 12.2 CREATING ICONS FOR THE TESTAPP1 APPLICATION.

I'll click on the **New** button in the dialog box that appears. If I had any icons in the resource file already, they'd show up in the dialog box list. That dialog box is shown in Figure 12.3.

After the dialog box is dismissed, the icon editor opens. For the TestApp1 application, I've created an icon that shows a library document being moved into the generic diamond application icon. To create the other icons shown in Figure 12.4, I simply dragged the small 'ICN#' straight down into the 'icl8' box. Then, I added some shading to the icon using the tools from the far left of the icon editor window. I repeated those steps to create the 'icl4' icon as well.

After closing the icon editor the 'BNDL' resource now looked like the one pictured in Figure 12.5.

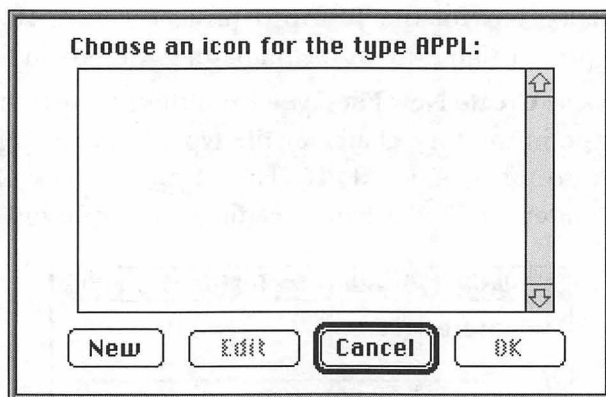


FIGURE 12.3 If a resource file has icons in it, ResEdit allows you to choose from them when creating a new icon.

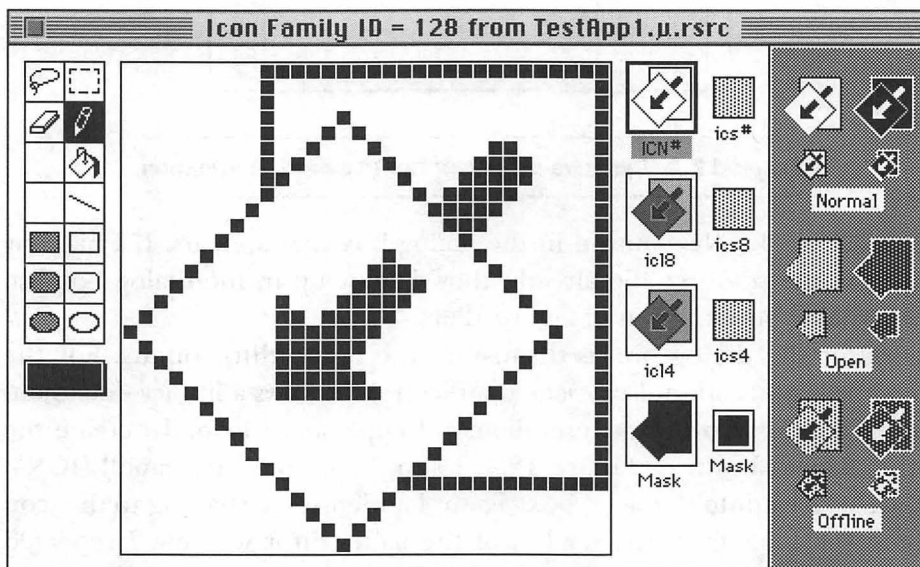


FIGURE 12.4 EDITING THE APPLICATION ICON IN ResEdit's ICON EDITOR.

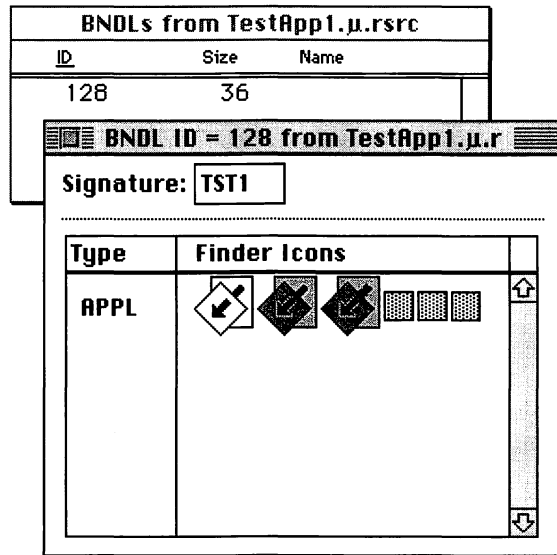


FIGURE 12.5 THE FAMILY OF THREE APPLICATION ICONS.

Adding an Icon to the Library

There are two ways to add an icon to an import library. You can add a 'BNDL' resource to the resource file of the import library, or you can add to the 'BNDL' in the application resource file. If you're creating a set of libraries that will be used as tools with a single application, you'll want to use the second method. If you're using Apple events—as you will be later in this chapter—this will have the side benefit of causing a double-click on an import library to launch the application and run the import library code.

By giving each import library a creator that is the same as the application that will use it, you can create a single 'BNDL' resource in the application to assign icons to each library. You'll see how that happens in this section.

Applications often have owned documents to which they assign their own icon. Figure 12.6 shows the 'BNDL' resource from the Microsoft

Programming the PowerPC

Word application. Besides the icon for the application itself, the 'BNDL' defines icons for documents that the word processor creates or uses.

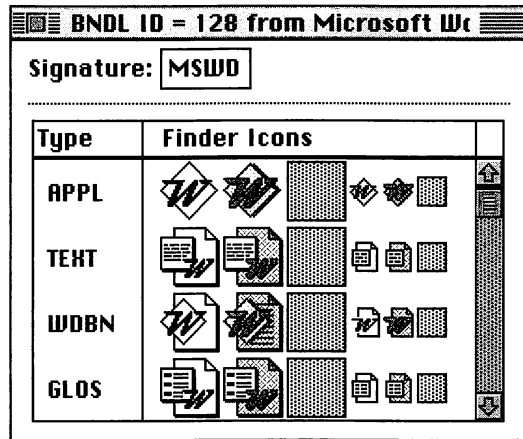


FIGURE 12.6 THE 'BNDL' RESOURCE FOR MICROSOFT WORD.

To create an icon for the CompanyInfo import library, I repeated the procedure I used for the application icon. As you read over those steps, recall that I gave the CompanyInfo library a signature of 'TST1' and a type of 'dLIB':

1. Open the 'BNDL' resource in the TestApp1 resource file.
2. Select **Create New File Type** from the Resource menu, hit the **tab** key, and type in the four character file type—'dLIB.'
3. Double-click on any of the shaded boxes under the Finder Icons heading.
4. Click on the **New** button in the dialog box that appears.
5. Create a new family of icons in the icon editor.

When I completed the above steps, the 'BNDL' resource looked like the one shown in Figure 12.7.

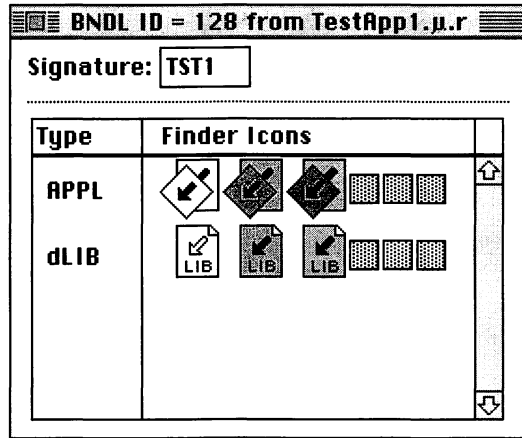


FIGURE 12.7 THE APPLICATION AND IMPORT LIBRARY ICONS FOR THE TESTAPP1 APPLICATION



However, the CompanyInfo library is created from a project and a resource file that are separate from the TestApp1 project and resource file. So how will the CompanyInfo library—which currently has a generic document icon—take on the icon created in the TestApp1.u.rsrc file?

Because I've set the creator of the library to be the same creator as the TestApp1 application (TST1). Thus, the Finder views the CompanyInfo library as a document owned by TestApp1—even though it is a shared library that can be used by other applications.

When your work on the 'BNDL' resource is complete, you'll notice that five new resource types have been added to the existing 'vers' and 'BNDL' types. The resource file is shown in Figure 12.8.

So far you've only changed the resource file that the project uses—you haven't changed the TestApp1 application. You'll want to launch CodeWarrior and make a new TestApp1. If you don't see the new icons on the application and the library after quitting CodeWarrior, restart your

Programming the PowerPC

Macintosh. If that doesn't work, rebuild the desktop to force the Finder to recognize the new icons. Rebuild the desktop by holding the **Command** and **Option** keys down while you restart your computer. When the new icons do show up, they'll look like the ones pictured in Figure 12.9.

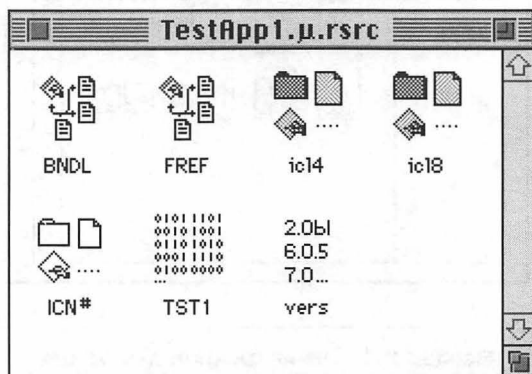


FIGURE 12.8 THE RESOURCES THAT MAKE UP THE TESTAPP1 RESOURCE FILE.

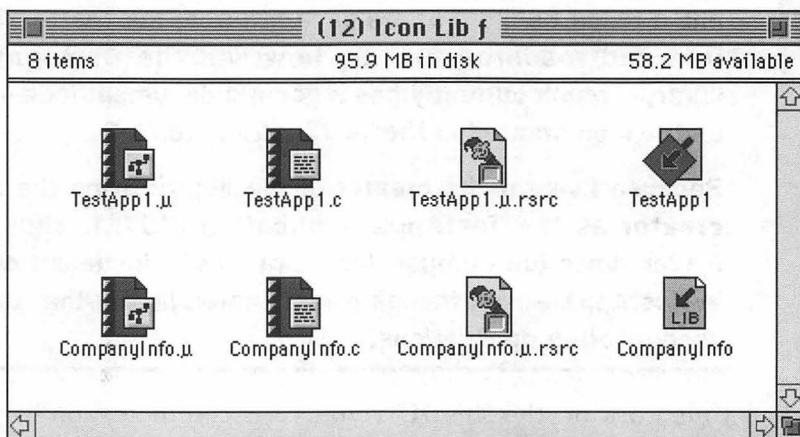


FIGURE 12.9 THE FOLDER THAT HOLDS BOTH THE APPLICATION AND IMPORT LIBRARY PROJECTS.

Now that the Finder recognizes a file with a creator of 'TST1' and a type of 'dLIB,' any file that has this creator and type will get the new icon. If I

created two additional libraries, and used the CodeWarrior preferences dialog box to set the creator to 'TST1' and the type to 'dLIB,' I'd have the results shown in Figure 12.10.

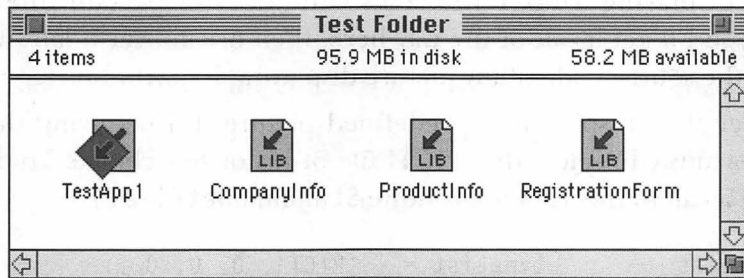


FIGURE 12.10 ANY IMPORT LIBRARY THAT IS BUILT WITH A CREATOR OF 'TST1' AND A TYPE OF 'dLIB' WILL HAVE THE SAME ICON.



NOTE

Of course, I'd have to modify the TestApp1 source code so that the application would actually do something with the libraries. But even without doing that, the point that the libraries all get the same icon still applies.

A SECOND LIBRARY EXAMPLE

The CompanyInfo library served as a good introduction to how an application makes use of an import library. But its simplicity leaves a few questions unanswered. Most importantly, once an import library is loaded, how does the application code interact with the library code on a continuous basis? This section will answer that question. In doing so, I'll develop a library named PICTchooser that performs the very useful act of displaying a standard get file dialog box that allows the user to open and display any existing 'PICT' file.

Opening a PICT File

Before discussing the next example library, I'll cover the details of opening and displaying a PICT file. This code doesn't pertain directly to libraries, so I'll get it out of the way here, then brush over it later when I use it in the source code of my picture-displaying import library.

Rather than displaying a predefined picture, I'll be giving the user the opportunity to select the 'PICT' file of his or her choice. To do that I'll make a call to the Toolbox routine `StandardGetFile()`:

```
SFTypelist      typeList = { 'PICT', 0, 0, 0 };
StandardFileReply  reply;

StandardGetFile( nil, 1, typeList, &reply );
```

The first three parameters to `StandardGetFile()` tell the function which types of files to display in the dialog box's list. The dialog box will display only the type or types you specify, masking out all other file types.

If you want to specify more than four different file types to display, you'll need to pass a pointer to a filter function. Since I'm only displaying one file type, `nil` will suffice for this first parameter. The second parameter tells how many types of files to list—a value of 1 is used here. The third parameter gives the types of files to list. This parameter is a variable of `SFTypelist` type, which is an array of four file types. Each type should be a four character file type surrounded by single quotes. Empty array elements should simply be assigned a value of 0. My declaration of the `SFTypelist` variable includes a definition of the one file type to be displayed:

```
SFTypelist  typeList = { 'PICT', 0, 0, 0 };
```

The final parameter to `StandardGetFile()` is a pointer to a reply structure. The `StandardFileReply` data structure members will be filled in for you by the Toolbox. You'll use the `sfFile` member when you open the picture file:

```
short pict_ref_num = 0;

FSpOpenDF( &reply.sfFile, fsRdPerm, &pict_ref_num );
```

The Toolbox routine `FSpOpenDF()` opens the data fork of the file whose `FSSpec` appears as the first parameter. The data fork of a ‘PICT’ file holds the picture information, so this is exactly what I want to work with. The second parameter to `FSpOpenDF()` is a permission level. The user won’t be given the opportunity to alter, or write, to the picture. Instead, the user will just be allowed to view or read it. So the constant `fsRdPerm` works here.

After opening the file’s data fork, `FSpOpenDF()` returns a file reference number to the program. You’ll use that reference number to gain access to the file. Namely, you’ll use it in calls to the Toolbox functions `GetEOF()` and `SetFPos()`:

```
long file_length;

GetEOF( pict_ref_num, &file_length );
SetFPos( pict_ref_num, fsFromMark, 512 );
```

The `GetEOF()` function returns the logical end-of-file for the file specified in the first parameter. The logical end-of-file will yield the actual number of bytes of data that the picture occupies—not the physical size allocated for the file, which may be a greater value. I’ll save this value in the long variable `file_length`.

All ‘PICT’ files have a 512 byte header that holds data unrelated to the picture data itself. I’ll use a call to `SetFPos()` to position the file mark just past this header information. The file mark tells the File Manager where to begin writing or, in my case, reading.

Now, I’m just about ready to read in the file information. First I’ll determine the actual number of bytes of data that are devoted to the picture by subtracting the 512 byte header from the total file length. Then I’ll use a call to `NewHandle()` to allocate some memory to hold the picture data:

```
Size pict_size;
```

Programming the PowerPC

```
Handle temp_handle = nil;

pict_size = file_length - 512;

temp_handle = NewHandle( pict_size );
```

Finally, it's time to read in the data from the open 'PICT' file and store it in the allocated memory. Since a handle is relocatable, I'll take the precaution of locking it in place while a call to `FSRead()` reads in the data:

```
HLock(temp_handle);
    FSRead(pict_ref_num, &pict_size, *temp_handle);
HUnlock(temp_handle);
```

You'll notice that I named the handle that holds the picture data `temp_handle`. That provides a hint that there's one step left. The Toolbox uses a `PicHandle` when working with a picture—not a generic handle. So I'll declare a `PicHandle` variable and typecast the generic handle to that type:

```
PicHandle My_Picture;

My_Picture = ( PicHandle )temp_handle;
```

To display the picture I'd open a window and make a call to `DrawPicture()`. I won't do that now, however. Instead, I'll move on to the particulars of the new example library. There you'll see both the standard get file code and the code that draws and updates the picture.

The Initialization Routine

Because a library's initialization routine executes just one time, this routine serves as the perfect place to place the standard get file code developed in the previous section. I'll assume that an application that uses this library includes a menu item named something like `Open Picture`. In response to that item being selected, the application will load the picture-viewing library. The loading of the library triggers the execution of

Chapter 12 More Import Libraries

the library's initialization routine. So to the user, the loading of the library results in the dialog box shown in Figure 12.11.

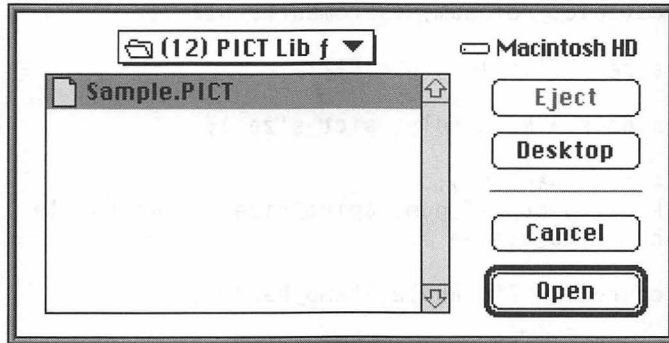


FIGURE 12.11 THE RESULT OF RUNNING THE INITIALIZATION ROUTINE OF THE PICTCHOOSER IMPORT LIBRARY.

I've included a single sample 'PICT' file on the disk that accompanies this book. You may very well have others on your hard drive. If you do, you'll be able to use the pop-up menu in the dialog box to locate them.

The source code for the library's initialization routine appears below. Since the code that makes up the function was developed in the previous section, I'll forego the usual walk-through. Instead, I'll just note that even though the routine doesn't make use of the `InitBlockPtr`, it still gets passed in as a matter of form.

```
//+++++++ initialization routine +++++++  
  
OSErr My_Initialize_Routine( InitBlockPtr init_block_ptr )  
{  
    SFTypelist      typeList = { 'PICT', 0, 0, 0 };  
    StandardFileReply reply;  
    short           pict_ref_num = 0;  
    long            file_length;  
    Size            pict_size;  
    Handle          temp_handle = nil;  
  
    StandardGetFile( nil, 1, typeList, &reply );
```

Programming the PowerPC

```
FSOpenDF( &reply.sfFile, fsRdPerm, &pict_ref_num );

GetEOF( pict_ref_num, &file_length );
SetFPos( pict_ref_num, fsFromMark, 512 );

pict_size = file_length - 512;

temp_handle = NewHandle( pict_size );

HLock( temp_handle );
    FSRead( pict_ref_num, &pict_size, *temp_handle );
HUnlock( temp_handle );

My_Picture = ( PicHandle )temp_handle;

return ( noErr );
}
```

You'll notice that the variable `My_Picture` isn't declared locally in the routine. That's because you'll need to use this variable later on to initially draw, and later update, the picture. Instead, the variable is declared global to the library. I've added its declaration to the top of the library, just after the inclusion of the `FragLoad.h` universal header file. That makes the variable accessible by any routines in the library—but not by any routines in an application fragment that makes use of the library.

```
//+++++ include directives ++++++

#include <FragLoad.h>

//+++++ global variables ++++++

PicHandle      My_Picture = nil;
```



Important: Error-checking is an important part of programming. You may have noticed that there's precious little of it in the initialization routine. This book deals with PowerPC programming—not with files and error handling. So for the sake of brevity, error-checking is slight. Typically, you'll want to examine the error value that Toolbox functions

return to guarantee that they've executed successfully. For example, I've written the call to `FSpOpenDF()` as follows:

```
FSpOpenDF( &reply.sfFile, fsRdPerm, &pict_ref_num );
```

You might instead want to write it like this:

```
OSErr error;

error = FSpOpenDF( &reply.sfFile, fsRdPerm,
                  &pict_ref_num );
if ( error != noErr )
{
    Post_Error_Message( FILE_OPEN_ERR );
    ExitToShell();
}
```

To see if a Toolbox function returns an error code, check the universal header files or the Apple Inside Macintosh series of books.

The Termination Routine

The CompanyInfo library had no termination routine, since there was nothing to clean up—that's the typical role of this function. The PICTchooser library, on the other hand, is the perfect example of a library that can use a termination routine. This library has a global variable named `My_Picture` that serves as a handle to the picture data. Since this library allocated the memory for the data, it makes sense that this library should also be responsible for freeing up this memory when it is no longer needed.

How do I know just when the memory for the picture is no longer needed? When the library is unloaded by the application, I'll assume that the user is through with the picture. Unloading the library triggers a call to the library's termination routine, so I'll just make a call to the Toolbox function `KillPicture()` at that time.

Programming the PowerPC

```
//+++++ termination routine ++++++

void My_Terminate_Routine( void )
{
    if ( My_Picture != nil )
        KillPicture( My_Picture );
}
```

The Main Routine

The main routine of a library is the one special routine that is explicitly called by another fragment. As such, any fragment that makes use of a library must be aware of the format of the library's main routine. For the PICTchooser example, the main function accepts two parameters. The calling fragment will have to be aware of that—as you'll see later in this chapter when I develop a simple test application that uses PICTchooser.

There is no one set format for a library's main routine—its return value and parameters will depend on the function's purpose. The purpose of PICTchooser is to open and display a picture. The initialization routine handles the opening of the picture. I'll have the main routine handle its display.

True, I could have displayed the picture in the initialization routine. But a picture displayed in a window will need updating. So it makes more sense to draw the picture from a routine that can be called repeatedly by the application fragment. Here's a look at the complete main routine:

```
//+++++ main routine ++++++

Boolean My_Main_Routine( EventRecord *the_evt,
                        WindowPtr    front_wind )
{
    Boolean evt_handled = false;

    if (the_evt->what == updateEvt)
    {
        BeginUpdate( front_wind );
        SetPort( front_wind );
        DrawPicture( My_Picture, &front_wind->portRect );
        EndUpdate( front_wind );
        evt_handled = true;
    }
}
```

```
    }  
    return ( evt_handled );  
}
```

The main routine accepts an `EventRecord` and a `WindowPtr` as its two parameters, and returns a `Boolean` value. The assumption is that when an event occurs, the application fragment will call `PICTchooser`'s main routine to see if the event involved the picture. The application will pass the main routine a pointer to its event record, as well as a pointer to the window that was involved in the event. The main routine will check to see if the event was update-related. If it was, it will draw the picture into the port of the window whose `WindowPtr` was passed in `front_wind`.

If the main routine did in fact redraw the picture, the local variable `evt_handled` is set to `true`. If the main routine didn't handle the event, this variable will have its initialized value of `false`. In either case, the value of this variable serves as the function's return value. When the function has completed, the calling application will be informed as to whether or not the event has been handled.



Forcing the picture to fit the window's `portRect` is just a quick and dirty way of displaying it. A better way would be to start with the window hidden. Then determine the picture's size, resize the window to those same dimensions, show the window, and finally, display the picture.

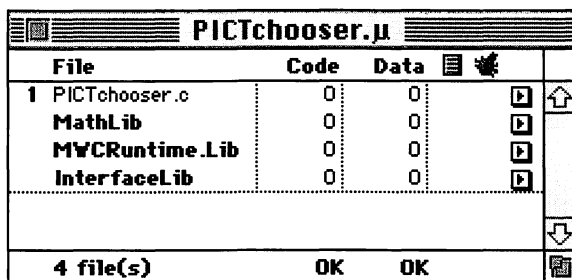
By looking at `My_Main_Routine()` you can see that the main routine makes certain assumptions about the calling application. For instance, it assumes there is already a window open. Later in this chapter I'll develop a test application that demonstrates the interaction between an application fragment and the main routine of an import library fragment.

Using CodeWarrior to Build the Library

In Chapter 11 you saw how to use CodeWarrior to build a the `CompanyInfo` library. For the `PICTchooser` library the process will be the same.

Programming the PowerPC

PICTchooser has no required resources. So its resource file will hold just the 'vers' resource that is commonly found in an import library. The Metrowerks C/C++ PPC project window—shown in Figure 12.12—holds the source code file and the three libraries that are added to all CodeWarrior projects.



**FIGURE 12.12 THE CODEWARRIOR PROJECT WINDOW
FOR THE PICTCHOOSER IMPORT LIBRARY.**

Before building the library, I'll make a few changes to the project using the preferences dialog box. In Figure 12.13 you can see that I've used the Project panel of the dialog box to make sure that the project type is set to shared library, and the creator and type are set to the desired four character strings.



NOTE

Remember, the names you choose for the creator and for the type of a shared library don't in any way limit which applications can make use of the shared library. You're choosing names for the purpose of giving the library a particular icon. If you want the shared library to have a particular icon associated with it, then you'll set the library's creator to that of an application that has a 'BNDL' resource that includes the type you give the library. For example, the TestApp1 has a creator of 'TST1,' and it has a 'BNDL' resource that defines the icons for 'APPL' and 'dLIB' type fragments. By setting the creator and type of a shared

library to 'TST1' and 'dLIB,' respectively, you're telling the Finder to look at how the TestApp1 application defines the icon of a 'dLIB' fragment.

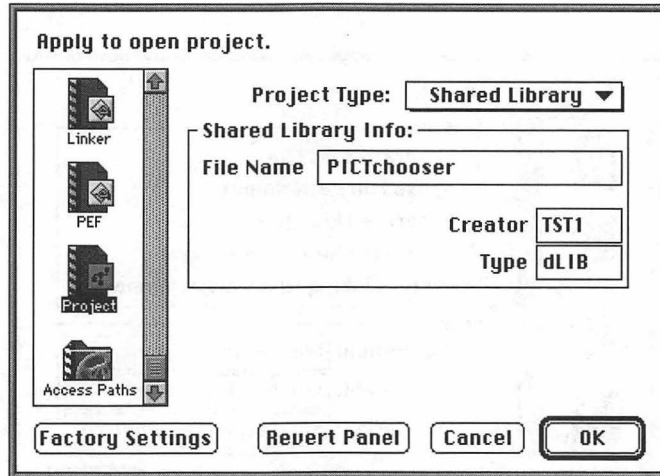


FIGURE 12.13 THE PROJECT PANEL SETTINGS FOR THE PICTCHOOSER IMPORT LIBRARY.

Next, I used the Linker panel of the preferences dialog box to set the library's three entry routines. As shown in Figure 12.14, these three names match the names of the three special routines I developed earlier in this chapter for the PICTchooser library.

After dismissing the preferences dialog box, I selected **Make** from the Project menu to build the library. In the next section I'll test the library out by loading it from an application fragment.

MODIFYING THE TESTAPP2 APPLICATION

The TestApp2 application developed in Chapter 11 is a good vehicle for testing import libraries—all it takes is a menu selection to load and execute one. To test the PICTchooser library I'll again use TestApp2—with a

Programming the PowerPC

few modifications. First, I'll need to change the resource file by adding a 'WIND' resource and a second menu item to the project's Utilities 'MENU' resource. Next, I'll need to change some of the source code to allow the application to interact on a continuous basis with the import library.

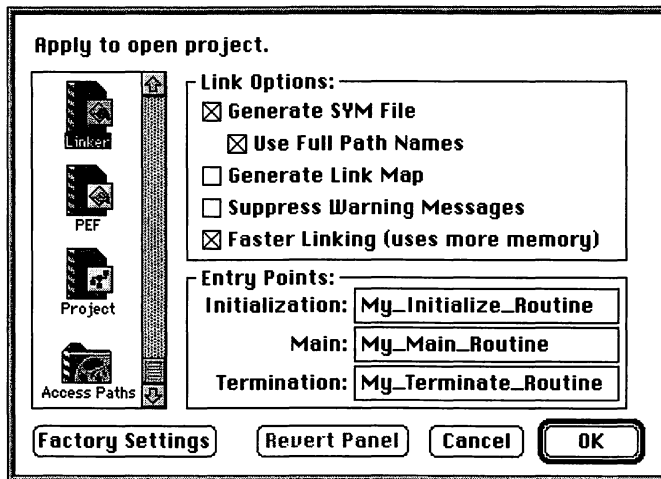


FIGURE 12.14 THE LINKER PANEL SETTINGS FOR THE PICTCHOOSER IMPORT LIBRARY.

Changes to the TestApp2 Resources

To allow TestApp2 to load either of my two import libraries, I'll add a new "Open Picture" menu item to one of the 'MENU' resources. Figure 12.15 shows how the Utilities 'MENU' resource has changed.

The PICTchooser library allows the user to select a 'PICT' file to open, and then the library displays the picture in a window. The library makes the assumption that the calling fragment—in this case TestApp2—opens a window to draw into. For that reason I've added a 'WIND' resource to the TestApp2.p.rsrc file. Because the import library will shrink or expand the picture to fit the size of the window it is drawing to, the window's size isn't critical. Figure 12.16 shows the 'WIND' resource I've added.

Chapter 12 More Import Libraries

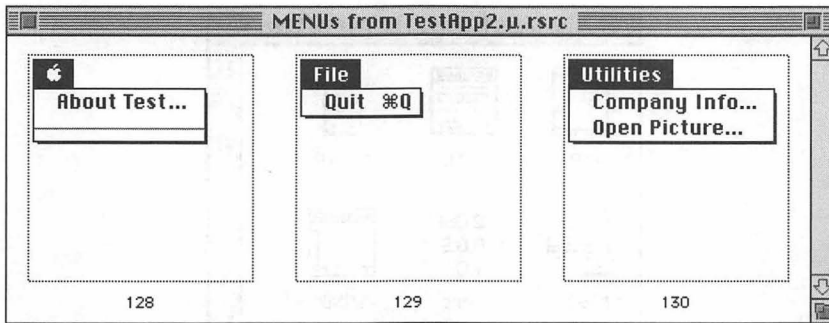


FIGURE 12.15 THE 'MENU' RESOURCES FOR THE RESOURCE FILE FOR THE MODIFIED TESTAPP2.

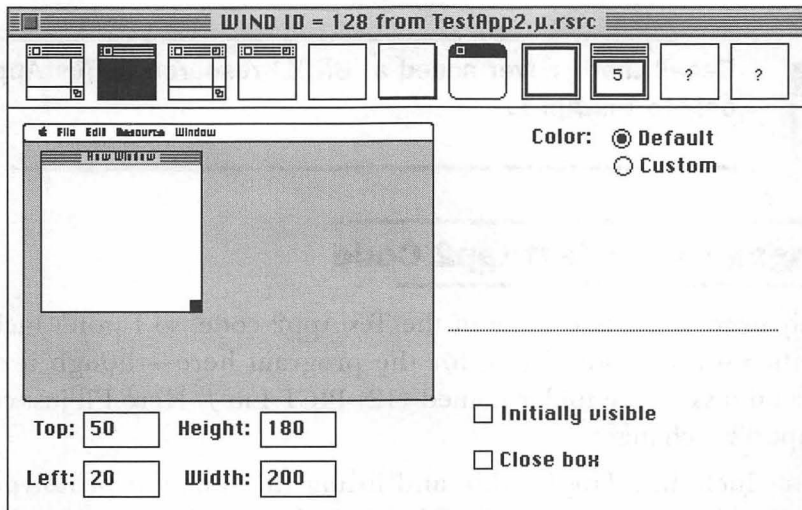


FIGURE 12.16 THE 'WIND' RESOURCE FOR THE RESOURCE FILE FOR THE MODIFIED TESTAPP2.

After the above changes, the TestApp2.p.rsrc file will look like the one pictured in Figure 12.17.

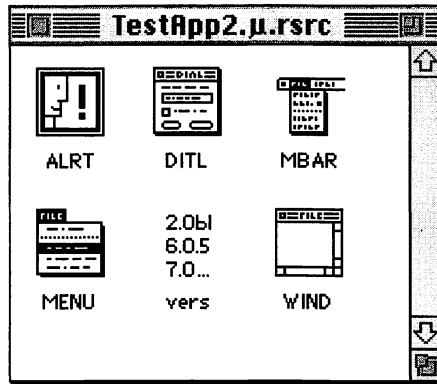


FIGURE 12.17 THE RESOURCES THAT MAKE UP THE RESOURCE FILE FOR THE MODIFIED **TESTAPP2**.



NOTE

Recall that I never added a 'BNDL' resource to TestApp2—only to TestApp1.

Changes to the TestApp2 Code

I'll only need to change some of the TestApp2 code, so I won't include the entire source code listing for the program here—though it does appear on disk in the folder named (12) PICT Lib *f*. Here I'll just cover the important changes.

After including FragLoad.h, and listing the function prototypes, I add the program's #define directives. All the old TestApp2 #defines are here, and two additions. I've added the name of the new import library right after the #define for the CompanyInfo library:

```
#define CO_INFO_LIB_STR      "\\pCompanyInfo"
#define GET_PICT_LIB_STR    "\\pPICTchooser"
```

Now that my program has a window, I've added a #define directive for that window's 'WIND' resource:

```
#define            MAIN_WIND_ID            128
```

The PICTchooser library doesn't open a window to hold the picture that it opens—it relies on the calling fragment to do that. So my test application will declare a WindowPtr variable, then make a call to GetNewCWindow() in main():

```
WindowPtr    My_Window = nil;

void   main( void )
{
    Initialize_Toolbox();
    Set_Up_Menu_Bar();

    My_Window = GetNewCWindow( MAIN_WIND_ID, NULL,
                              (WindowPtr)-1L );

    SetPort( My_Window );
    HideWindow( My_Window );

    Main_Event_Loop();
}
```

The next addition to TestApp2 may be the trickiest. Recall from Chapter 11 that the Code Fragment Manager routine GetDiskFragment() uses its seventh parameter to return a pointer to the main routine of the import library fragment that it loads. Figure 12.18 highlights this point.

In past examples I wasn't too concerned about the pointer Lib_Main_Ptr—the CompanyInfo library has no main routine. Now, because the PICTchooser library *does* have a main routine, I'll need to put a little forethought into things. Specifically, I'll need to add an interface to the TestApp2 source code so that it knows the format of the PICTchooser main routine. Looking back at the PICTchooser source code, I see that I've defined the main routine to look like this:

```
Boolean    My_Main_Routine( EventRecord *the_evt,
                           WindowPtr    front_wind )
{
    // function body
}
```

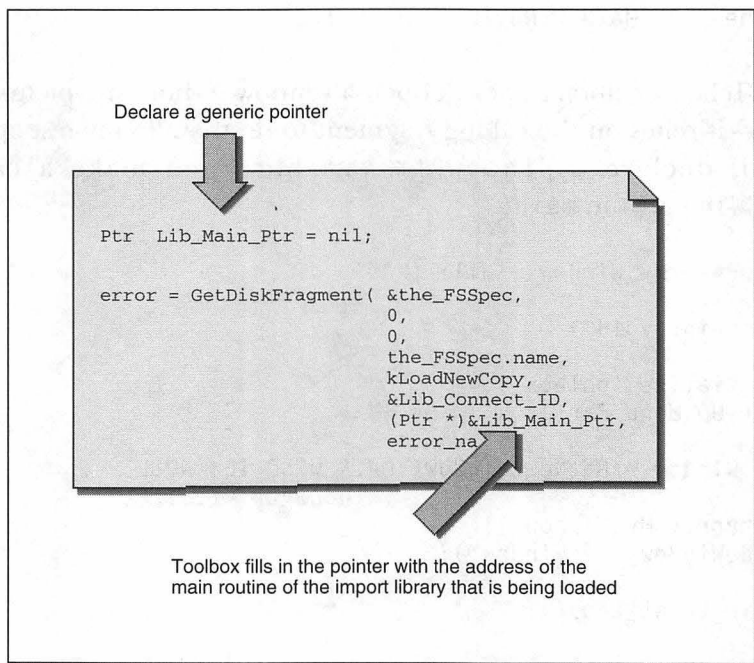


FIGURE 12.18 GetDiskFragment() RETURNS
A POINTER TO AN IMPORT LIBRARIES MAIN ROUTINE.

The PICTchooser main routine has two parameters and a Boolean return type. If I want to define a pointer to this main routine, I could do so as follows:

```
typedef Boolean (*MainRoutinePtr) ( EventRecord *, WindowPtr );
```

The above definition creates a type named MainRoutinePtr that is a pointer to a function that returns a Boolean, and has a pointer to an EventRecord and a pointer to a window as its two parameters. Now, instead of declaring Lib_Main_Ptr to be a generic Ptr type, I'll declare it to be a pointer to the main routine of the PICTchooser library:

```
MainRoutinePtr Lib_Main_Ptr = nil;
```

My test program allows the user to load the PICTchooser library at any time—a selection from the Utilities menu does that. That means that at any given time the program may have an import library with a main routine loaded. So I'll need to make a few changes to the way the program's main event loop handles events. After grabbing hold of an event with a call to `WaitNextEvent()`, the program should check to see if the event should be handled by the import library. Here's how that can be accomplished:

```
void Main_Event_Loop( void )
{
    EventRecord  the_event;
    Boolean      event_handled_by_lib;
    char         the_key;
    WindowPtr    the_window;
    long         menu_choice;

    while ( All_Done == false )
    {
        WaitNextEvent( everyEvent, &the_event, 15L, nil );

        event_handled_by_lib = false;

        if ( Lib_Main_Ptr != nil )
        {
            the_window = FrontWindow();
            event_handled_by_lib = Lib_Main_Ptr( &the_event,
                                                the_window );
        }

        if ( event_handled_by_lib == false )
        {
            // handle as a normal event
        }
    }
}
```

If a library is loaded, `Lib_Main_Ptr` will not be `nil`—it will hold the address of that library's main routine. If that's the case, the library's main routine should be called to be given the opportunity to handle the event. The above snippet makes a call to `FrontWindow()` to determine which window is at the front. It then passes the `WindowPtr` returned by `FrontWindow()`, along with a pointer to the `EventRecord`, to the import library's main routine.

Programming the PowerPC

The above version of `Main_Event_Loop()` is written such that if an import library is loaded, every pass through the while loop will result in a call to the import library's main routine. Assuming that the `PICTchooser` library is loaded, its main routine would be called. Here's a reminder of what takes place in that routine:

```
Boolean  My_Main_Routine( EventRecord *the_evt,
                          WindowPtr    front_wind )
{
    Boolean  evt_handled = false;

    if (the_evt->what == updateEvt)
    {
        BeginUpdate( front_wind );
        SetPort( front_wind );
        DrawPicture( My_Picture, &front_wind->portRect );
        EndUpdate( front_wind );
        evt_handled = true;
    }
    return ( evt_handled );
}
```

The `PICTchooser` main routine will update the front window by redrawing the picture to it. Note that `My_Picture`, which is global to the import library, but not declared in the test application, has retained its value.

When `My_Main_Routine()` has executed, it will return a Boolean value to the calling routine. If the event wasn't handled, a value of `false` is returned. If the event was handled, a value of `true` is sent back. Back in the test application, this returned value is examined to determine if the event still needs to be handled. If it does, then the application's `Main_Event_Loop()` will handle it:

```
if ( event_handled_by_lib == false )
{
    // handle as a normal event
}
```

Here, in its entirety, is the test application's `Main_Event_Loop()` function:

Chapter 12 More Import Libraries

```
void Main_Event_Loop( void )
{
    EventRecord  the_event;
    Boolean      event_handled_by_lib;
    char         the_key;
    WindowPtr    the_window;
    long         menu_choice;

    while ( All_Done == false )
    {
        WaitNextEvent( everyEvent, &the_event, 15L, nil );

        event_handled_by_lib = false;

        if ( Lib_Main_Ptr != nil )
        {
            the_window = FrontWindow();
            event_handled_by_lib = Lib_Main_Ptr( &the_event,
                                                the_window );
        }

        if ( event_handled_by_lib == false )
        {
            switch (the_event.what)
            {
                case mouseDown:
                    Handle_Mouse_Down( the_event );
                    break;

                case keyDown:
                    the_key = (the_event.message & charCodeMask);
                    if ( ( the_event.modifiers & cmdKey ) != 0 )
                    {
                        menu_choice = MenuKey( the_key );
                        Handle_Menu_Choice( menu_choice );
                    }
                    break;

                case updateEvt:
                    the_window = (WindowPtr)the_event.message;
                    BeginUpdate( the_window );
                    // update window as needed by the application
                    EndUpdate( the_window );
                    break;
            }
        }
    }
}
```

Programming the PowerPC

The only other change to TestApp2 occurs in the `Handle_Utility_Choice()` function. This routine gets called in response to a click in the Utilities menu. Since TestApp2 now has two menu items, I've added a second case label to the function's switch statement. A click on the **Open Picture** menu item results in the hidden window being shown, and the PICTchooser library being loaded.

```
void Handle_Utility_Choice( short the_item )
{
    FSSpec the_FSSpec;

    switch ( the_item )
    {
        case CO_INFO_ITEM:
            HideWindow( My_Window );
            the_FSSpec = Get_File_Spec( CO_INFO_LIB_STR );
            Load_Library( the_FSSpec );
            break;

        case OPEN_PICT_ITEM:
            ShowWindow( My_Window );
            the_FSSpec = Get_File_Spec( GET_PICT_LIB_STR );
            Load_Library( the_FSSpec );
            break;
    }
}
```

A Last Word on the Main Routine

It's important to keep in mind that a library's main routine has no standard interface. For my library, passing a pointer to the event record and a pointer to the front window sufficed. A different library may require other information from the calling fragment. For instance, if my main routine needed to access the QuickDraw global variables, I'd want to pass those along as a parameter. If I wanted to draw a black rectangle in the middle of my picture, I could change the `My_Main_Routine()` to look like this:

```
Boolean My_Main_Routine( EventRecord *the_evt,
                        WindowPtr front_wind,
                        QDGlobals *qd_ptr )
```

```
{
    Boolean    evt_handled = false;
    QDGlobals  temp = *qd_ptr;
    Rect       the_rect;

    if (the_evt->what == updateEvt)
    {
        BeginUpdate( front_wind );
        SetPort( front_wind );
        DrawPicture( My_Picture, &front_wind->portRect );
        SetRect( &the_rect, 10, 10, 50, 50 );
        FillRect( &the_rect, &temp.black );
        EndUpdate( front_wind );
        evt_handled = true;
    }
    return ( evt_handled );
}
```

This new version of the main routine now has three parameters rather than two. That means that in the test application the interface to `My_Main_Routine()` needs to be changed as well:

```
typedef Boolean  (*MainRoutinePtr) ( EventRecord *,
                                    WindowPtr,
                                    QDGlobals * );
```

And, any calls to the library's main routine will need to include that third parameter:

```
if ( Lib_Main_Ptr != nil )
{
    the_window = FrontWindow();
    event_handled_by_lib = Lib_Main_Ptr( &the_event,
                                         the_window, &qd );
}
```

If you're writing more than one library, you may want to consider giving the main routine of each library the same interface. That way your application can use a single line of code to access which ever main routine is currently open:

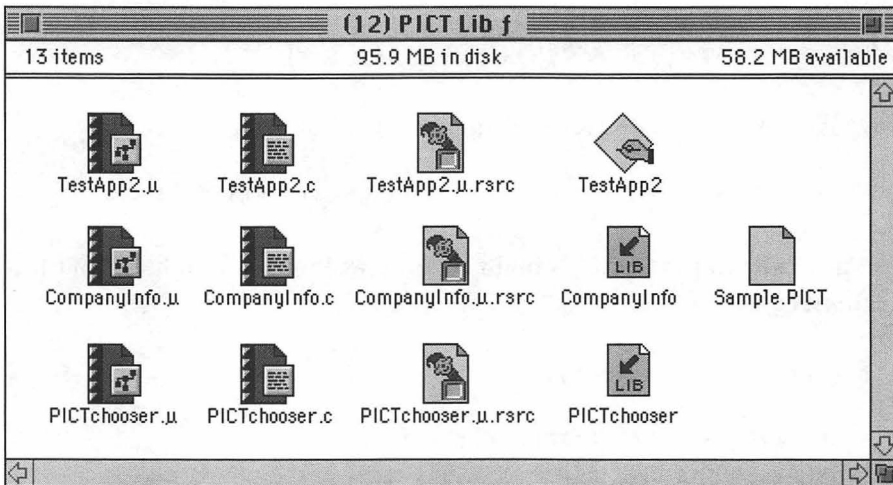
Programming the PowerPC

```
event_handled_by_lib = Lib_Main_Ptr( &the_event, the_window,  
                                     &qd );
```

Even if the main routine of one or more libraries doesn't make use of the QDGlobals variable `qd`, or one of the other parameters, it will still be accessible by the application via the above call to `Lib_Main_Ptr`.

Testing the PICTchooser Library

After making the necessary changes to the `TestApp2` resource file and source code file, build a new `TestApp2` application by selecting **Make** from CodeWarrior's Project menu. The (12) PICT Lib *f*, shown in Figure 12.19, contains new copies of all of the application and library files. The `CompanyInfo` project hasn't changed, so these files are all the same as previous versions.



**FIGURE 12.19 THE FOLDER THAT HOLDS THE
TEST APPLICATION AND IMPORT LIBRARY PROJECTS.**

To test the new library, I'll launch `TestApp2`. Selecting **Open Picture** from the Utilities menu displays an empty window and posts the standard get file dialog box. Figure 12.20 shows this.

Chapter 12 More Import Libraries

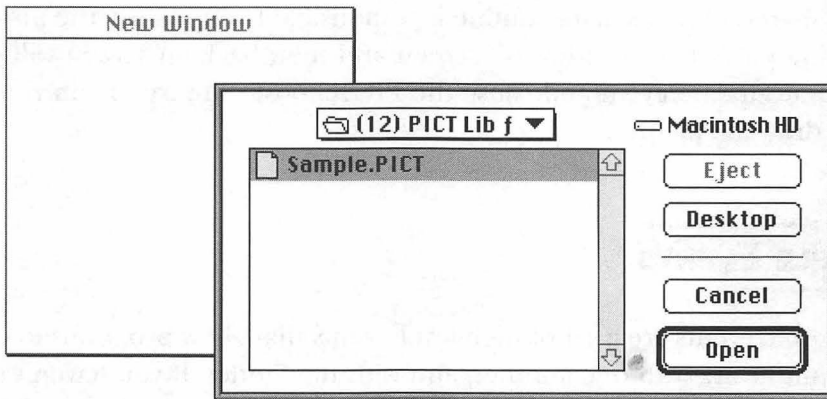


FIGURE 12.20 THE RESULT OF SELECTING OPEN PICTURE FROM THE TESTAPP2 UTILITIES MENU.

The TestApp2 program, by way of the code in the PICTchooser shared library, can open any 'PICT' file. There's one sample 'PICT' included on the disk—Figure 12.21 shows how the 'PICT' looks in the TestApp2 application. If you have any other 'PICT' files on your hard disk you'll be able to open them with TestApp2.



FIGURE 12.21 THE PICTCHOOSER LIBRARY WILL OPEN A 'PICT' FILE AND DRAW THE CONTENTS TO A WINDOW.

Programming the PowerPC

The shared library's main routine is responsible for updating the picture. Moving part of the window off screen and then back on screen will generate an update event and cause the PICTchooser library's main routine to redraw the picture.

APPLE EVENTS

Apple events are a set of high-level events that allow programs to communicate with one another, and with the Finder. If you haven't used Apple events in the past, you'll want to start using them now. That's because System 7.x is now the standard Macintosh operating system, and Apple defines four "required" Apple events that any program that runs under System 7.x should be able to respond to.

Introduction to Apple Events

Applications generally use Apple events to either request a service from another application, or to provide a service to another application. Typically that other application is the Finder. Which ever application initiates the event is said to be the *client application*, while the application that responds, or provides the requested service, is called the *server application*.

There are numerous Apple events, but there are only four that all System 7 applications are required to watch for: Open Documents, Open Application, Print Documents, and Quit Application. Each of these four required event types fall into the broader category of *core Apple events*.



The topic of Apple events is another one that is worthy of its own book. In this book, I'll concentrate on the Quit Application event and the Open Document event.

Responding to a Quit Application Apple Event

When you select **Restart** or **Shut Down** from the Special menu of the Finder, you expect any open applications to quit before your Mac powers down. In turn, the Finder will go to each application and terminate it. If, in the middle of this process, the Finder goes to an application but doesn't terminate it, you know that application doesn't support the Quit Application Apple event. The Finder attempts to send that application a request to quit, but the application doesn't recognize the Finder's efforts.

In this section you'll see how easy it is to give your application the power to recognize this one Apple event type. There are a few steps that all applications must follow if they are to work with Apple events. By giving your program the ability to respond to the Quit Application Apple event, you'll see the steps you'll need to follow to add other Apple event types as well.

To make your application aware of Apple events, add a case label to the switch statement in your program's main event loop. Besides checking for the standard event types such as `mouseDown`, `keyDown`, and `updateEvt`, you'll want to now watch for events of type `kHighLevelEvent`. Should `WaitNextEvent()` return an event of this type, your application will respond with a call to the Apple Event Manager function `AEProcessAppleEvent()`:

```
case kHighLevelEvent:
    AEProcessAppleEvent( &the_event );
```

Here's a look at the above code in the context of a typical application's main event loop:

```
void Main_Event_Loop()
{
    // local variable declarations

    while ( All_Done == false )
    {
        WaitNextEvent( everyEvent, &the_event, 15L, nil );
```

Programming the PowerPC

```
switch ( the_event.what )
{
    case mouseDown:
        // handle mouse click
        break;

    case updateEvt:
        // handle update event
        break;

    case kHighLevelEvent:
        AEProcessAppleEvent( &the_event );
}
}
```

The `AEProcessAppleEvent()` routine is a powerful function whose purpose is to identify the type of Apple event that is to be processed, and to begin processing that event. It starts the processing of the event by invoking an *Apple event handler*. An Apple event handler is a function that you provide. Each Apple event handler has a clearly defined purpose. It extracts data from the Apple event, handles the specific action that the event requests, and returns an error result code to indicate whether or not the event was successfully handled. How the functionality of the event handler routine is implemented is up to you. Here's how I implemented the event handler routine for a Quit Application Apple event:

```
pascal OSErr AE_Handle_Quit( AEDescList *apple_evt,
                             AEDescList *reply,
                             long        ref_con )
{
    All_Done = true;
    return noErr;
}
```

All my Quit Application event handler has to do to process a Quit Application Apple event is set the application's global variable `All_Done` to true. I'll make the assumption that this one line of code doesn't generate any kind of operating system error and return the global constant `noErr` to `AEProcessAppleEvent()`, the function that invoked `AE_Handle_Quit()`.

An event handler routine starts with the `pascal` keyword, and has a return type of `OSErr`. The event handler always has three parameters. The first parameter holds the Apple event to handle. Later in this chapter you'll see an example of how an event handler might use this information. If your event handler needs to return information to `AEProcessAppleEvent()`, it should fill in some of the fields of the second parameter. The last parameter is a reference value that your application will typically ignore.

There's one additional step that must be included in order for an application to work with Apple events. Each Apple event handler routine must be installed near the start of program execution. This is a necessary step that relates an Apple event type with the application-defined routine that will handle it. In my above example I've defined a routine named `AE_Handle_Quit()` to handle a Quit Application Apple event. But the Apple Event Manager routine `AEProcessAppleEvent()` has no way of knowing that this is the function that it should invoke in response to a Quit Application event. The installation of the event handler gives the Apple Event Manager this information. Here's the installer for my version of Quit Application:

```
AEInstallEventHandler( kCoreEventClass,  
                      kAEQuitApplication,  
                      NewAEEEventHandlerProc(AE_Handle_Quit),  
                      0,  
                      false );
```

The first parameter to `AEInstallEventHandler()` is the event class of the event to be handled. All four of the required Apple events are considered core events, so the installers for each of these four event handlers will have a first parameter `kCoreEventClass`.

The second parameter is an event ID that specifies which particular Apple event is to be handled. For the four required Apple events, those IDs are `kAEQuitApplication`, `kAEOpenApplication`, `kAEPrintDocuments`, and `kAEOpenDocuments`.

The third parameter to `AEInstallEventHandler()` is a pointer to the application-defined function that will handle this one Apple event.

Programming the PowerPC

You'll always use the `NewAEEEventHandlerProc()` function here to establish this pointer. Just provide the name of the event handler that you've created in your application.

The fourth parameter is a reference value that the Apple Event Manager will use each time it invokes the event handler function. You can safely use a value of 0 for this parameter.

The final parameter to `AEInstallEventHandler()` is a Boolean value that specifies in which *Apple event dispatch table* the handler should be added. An Apple event dispatch table provides the correlation between an Apple event and your application-defined event handler routine. Typically you'll provide a value of `false` here so that the Apple Event Manager adds the event handler to your application's own Apple event dispatch table, rather than to the system Apple event dispatch table that holds handlers that are available to all applications.

You'll want to install all your event handlers at application initialization time. Here I've added my one installer to the `Initialize_Toolbox()` function I include in all my applications:

```
void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();

    AEInstallEventHandler( kCoreEventClass, kAEQuitApplication,
                          NewAEEEventHandlerProc(AE_Handle_Quit),
                          0, false );
}
```

In the next section you'll see the source code for a complete application that handles all four of the required Apple event types. Until then, keep this summary of how to add Apple event handling to your application:

1. Add a case `kHighLevelEvent` label to the switch statement in your application's main event loop. Under the label, add a call to the Apple Event Manager routine `AEProcessAppleEvent()`.
2. Install each event handler near the top of your source code by calling `AEInstallEventHandler()` for each Apple event your application supports:

```
AEInstallEventHandler( kCoreEventClass, kAEQuitApplication,  
                     NewAEEEventHandlerProc(AE_Handle_Quit),  
                     0, false );
```

3. Define an event handler routine for each Apple event your application will support. This routine will typically have the same format as this Quit Application event handler:

```
pascal OSErr AE_Handle_Quit ( AEDescList *apple_evt,  
                             AEDescList *reply,  
                             long         ref_con )
```

ADDING APPLE EVENTS TO AN APPLICATION

The preceding section showed the steps you should perform in order to get your application to respond to Apple events. Now it's time for a specific example. In this section I'll modify the `TestApp2` code so that it works with Apple events. Since these changes will move my test application to a new plateau, I'll also give it a new name. Henceforth, the test application now will be known by the very clever name of `TestApp3`. You'll find the entire source code listing for `TestApp3` in the folder named (12) `AppleEvents Lib f`.

Apple events are useful, slick tools that give your applications that final professional polish. But how do they pertain to import libraries—the topic of this chapter? By making your application aware of the Open Documents Apple event type, you can allow a user of your application to launch that application by dragging an import library icon onto the application icon. Not only will that launch the application, it will execute the code in that import library.

Modifying the Main Event Loop

Getting your application to recognize Apple events is a three step process. The first step is to modify your application's main event loop by including a case `kHighLevelEvent` label to the switch statement in your application's main event loop. Then, under the label, add a call to the Apple Event Manager routine `AEProcessAppleEvent()`. Here's the complete event loop for the `TestApp3` application:

```
void Main_Event_Loop( void )
{
    EventRecord    the_event;
    Boolean        event_handled_by_lib;
    char           the_key;
    WindowPtr      the_window;
    long           menu_choice;

    while ( All_Done == false )
    {
        WaitNextEvent( everyEvent, &the_event, 15L, nil );

        event_handled_by_lib = false;

        if ( Lib_Main_Ptr != nil )
        {
            the_window = FrontWindow();
            event_handled_by_lib = Lib_Main_Ptr( &the_event,
                                                the_window );
        }

        if ( event_handled_by_lib == false )
        {
            switch ( the_event.what )
            {
                case mouseDown:
                    Handle_Mouse_Down( the_event );
                    break;

                case keyDown:
                    the_key = (the_event.message & charCodeMask);
                    if ( ( the_event.modifiers & cmdKey ) != 0 )
                    {
                        menu_choice = MenuKey( the_key );
                        Handle_Menu_Choice( menu_choice );
                    }
            }
        }
    }
}
```

```
        }
        break;

    case updateEvt:
        the_window = (WindowPtr)the_event.message;
        BeginUpdate( the_window );
        EndUpdate( the_window );
        break;

    // recognize and respond to Apple events!
    case kHighLevelEvent:
        AEPProcessAppleEvent( &the_event );

    }
}
}
```

Installing the Event Handlers

Each Apple event your application recognizes must have its own event handler routine, and those routines must be installed. My TestApp3 program will recognize the four required Apple event types, so I'll need to add four calls to `AEInstallEventHandler()`. Here's the new version of `Initialize_Toolbox()`, revised to install the four handlers:

```
void Initialize_Toolbox( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0 );
    InitCursor();

    AEInstallEventHandler( kCoreEventClass, kAEOpenApplication,
                          NewAEEEventHandlerProc
                          (AE_Handle_Open_App),
                          0, false );

    AEInstallEventHandler( kCoreEventClass, kAEOpenDocuments,
                          NewAEEEventHandlerProc
                          (AE_Handle_Open_Doc),
                          0, false );
}
```

Programming the PowerPC

```
AEInstallEventHandler( kCoreEventClass, kAEPrintDocuments,
                      NewAEEEventHandlerProc
                      (AE_Handle_Print_Doc),
                      0, false );

AEInstallEventHandler( kCoreEventClass, kAEQuitApplication,
                      NewAEEEventHandlerProc
                      (AE_Handle_Quit),
                      0, false );
}
```

Defining the Event Handlers

The complexity of an Apple event handler routine varies with the task that handler is to perform. You've already seen that the Quit Application event handler consists of nothing more than an assignment that toggles `All_Done` to true and a return statement that tells the calling routine that everything went all right:

```
pascal OSErr AE_Handle_Quit( AEDescList *apple_evt,
                             AEDescList *reply,
                             long        ref_con )
{
    All_Done = true;
    return noErr;
}
```

TestApp3 recognizes three other Apple events—but it only responds to one of them. When the user drags an import library icon onto the TestApp3 icon, the Finder—the client—will initiate an Open Document Apple event that will be serviced by TestApp3—the server. The next section describes in detail just how the Open Document event handler is implemented. TestApp3 has made provisions for the other two required Apple events—but doesn't really respond to them. Both an Open Application event and a Print Document event will result in an event handler being invoked. But I've left these two event handlers as nothing more than shells that can be filled in at a later time. The constant

`errAEventNotHandled` simply tells the Apple Event Manager that nothing was done in response to either event.

```
pascal OSErr AE_Handle_Open_App( AEDescList *apple_evt,
                                AEDescList *reply,
                                long         ref_con )
{
    return errAEventNotHandled;
}

pascal OSErr AE_Handle_Print_Doc( AEDescList *apple_evt,
                                AEDescList *reply,
                                long         ref_con )
{
    return errAEventNotHandled;
}
```



Is my handling of these two Apple event types a cop out? Perhaps. But again, keep in mind that this isn't a book about Apple events. At least now you're all set up to handle all four required Apple events. And in just a bit I'll cover the one event type that makes the most sense for working with import libraries—the Open Document Apple event. For a comprehensive look at Apple events, refer to *Inside Macintosh: Interapplication Communication*.

Defining the Open Document Event Handler

By having my `TestApp3` program recognize an Open Document Apple event, I provide it with the capability of performing the neat trick of launching itself and executing an import library when an import library icon is dragged onto the application icon. To get the `TestApp3` application to do that, I'll need to use a few Apple Event Manager functions in the Open Document event handler.

The `AEGetKeyDesc()` function is used by an event handler to extract information from an Apple event. Here's a typical call:

Programming the PowerPC

```
OSErr    error;
AEDesc   file_list_desc = { 'NULL', NULL };

error = AEGetKeyDesc( apple_evt,
                      keyDirectObject,
                      typeAEList,
                      &file_list_desc );
```

`AEGetKeyDesc()` accepts four parameters. The first is an `AEDescList`—a descriptor list that holds information about the Apple event. You can just pass the first parameter of the Open Document event handler for this `AEGetKeyDesc()` parameter:

```
pascal OSErr AE_Handle_Open_Doc( AEDescList *apple_evt,
                                  AEDescList *reply,
                                  long         ref_con )
```

The second parameter is a constant that is an `AEKeyword` that helps further identify information about the event. Use `keyDirectObject` here. The third parameter is a `DescType`. This is a four-character string that gives the Apple Event Manager still more information about the event. Pass the constant `typeAEList` here.

In exchange for the information supplied in the first three parameters, `AEGetKeyDesc()` will fill the last parameter with a descriptor record, or `AEDesc`. You'll use this `file_list_desc` variable in a call to another Apple Event Manager routine.

`AEGetNthPtr()` is used to get information from the `AEDesc` variable that was filled and returned by `AEGetKeyDesc()`. Though `AEGetNthPtr()` has a handful of parameters, there is only one that will be of interest to you—the sixth parameter. This parameter is a pointer to an `FSSpec` for the file involved in the Open Document Apple event—the import library that was dragged onto the application. Here's a call to `AEGetNthPtr()`:

```
FSSpec    file_spec;
OSErr     error;
AEKeyword  returned_keyword;
DescType   returned_type;
long       returned_size;
```

```
error = AEGetNthPtr( &file_list_desc,
                    1,
                    typeFSS,
                    &returned_keyword,
                    &returned_type,
                    (Ptr)&file_spec,
                    sizeof( file_spec ),
                    &returned_size );
```

The first parameter is the `AEDesc` variable returned by `AEGetKeyDesc()`. The second parameter is an index into a list of descriptor records. There's only one in the list, so this parameter is set to 1. The third parameter specifies what information I'm attempting to get from `file_list_desc`. Here, I want a `FSSpec`, so I use the constant `typeFSS`. The fourth and fifth parameters, `returned_keyword` and `returned_type`, return information that I won't be needing. The sixth parameter is what I'm looking for—a pointer to an `FSSpec` for the import library. The seventh and eighth parameters deal with the size of the returned data.

Now that I have an `FSSpec` for the import library, I can load the import library. A call to the application-defined routine `Load_Library()` handles that. After loading the import library code, I'll make a call to `AEDisposeDesc()` to dispose of the descriptor record that was declared in this routine and filled by `AEGetKeyDesc()`. Figure 12.22 summarizes the process of opening the dragged import library. The figure points out that `AEGetKeyDesc()` is called only to fill the `AEDesc` variable `file_list_desc`, and `AEGetNthPtr()` is called only to extract an `FSSpec` from the `file_list_desc` variable.

Below is the complete source code for an `AE_Handle_Open_Doc()` routine. Notice that most of the Apple Event Manager function parameters are constants or variables that get filled by the function calls—the only information you need to supply is found in the passed-in `AEDescList` variable `apple_evt`. That means the `AE_Handle_Open_Doc()` code will work—as is—for any import library—you shouldn't have to change the parameters to the two Apple Event Manager function calls in your own applications.

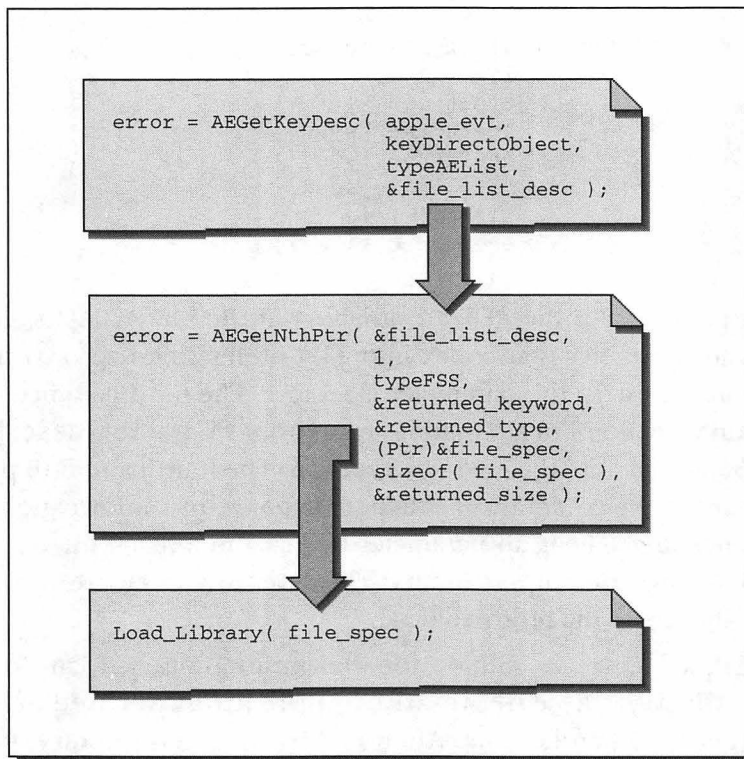


FIGURE 12.22 THE TWO APPLE EVENT MANAGER FUNCTION CALLS ARE MADE TO GET AN FSSPEC FOR THE IMPORT LIBRARY.

```
pascal OSErr AE_Handle_Open_Doc( AEDescList *apple_evt,
                                  AEDescList *reply,
                                  long         ref_con )
{
    AEDesc    file_list_desc = { 'NULL', NULL };
    FSSpec    file_spec;
    OSErr     error;
    AEKeyword  returned_keyword;
    DescType   returned_type;
    long       returned_size;

    error = AEGetKeyDesc( apple_evt, keyDirectObject,
                          typeAEList, &file_list_desc );
```

```
if ( error == noErr )
{
    error = AEGetNthPtr( &file_list_desc,
                        1,
                        typeFSS,
                        &returned_keyword,
                        &returned_type,
                        (Ptr)&file_spec,
                        sizeof( file_spec ),
                        &returned_size );

    if ( error == noErr )
        Load_Library( file_spec );
}

AEDisposeDesc( &file_list_desc );

return error;
}
```

Testing Apple Events

With the Apple event code added to the TestApp3 application, it's a simple matter to test things out. First build a TestApp3 application. You'll want to leave the creator as 'TST1' so that the application will be able to communicate with the libraries, which also have a creator of 'TST1.' Then, from the Finder, drag the CompanyInfo library icon onto the TestApp3 application icon. When you do, the TestApp3 application will launch. Not only that, but the CompanyInfo import library code will execute. Click on the CompanyInfo alert to dismiss it and end the program.

What about the PICTchooser library? If you built it with a creator of 'TST1', it too should be capable of launching TestApp3. If you drag the PICTchooser library icon onto the TestApp3 icon, the application will indeed launch. And, the standard get file dialog box will open. But you'll notice that there is no empty window open in the background. That's because the window is created in the application's `main()` routine, and then hidden with a call to `HideWindow()`. The window doesn't become visible until a call to `ShowWindow()` is made when the user selects **Open Picture** from the application's Utilities menu. When the application is launched via PICTchooser, the `ShowWindow()` call isn't made. The solu-

Programming the PowerPC

tion? Launching the application through an Apple event will execute the application's `main()` routine. So in `TestApp3`, comment out the `HideWindow()` call and add a call to `ShowWindow()`, as shown here:

```
void main( void )
{
    Initialize_Toolbox();
    Set_Up_Menu_Bar();

    My_Window = GetNewCWindow( MAIN_WIND_ID, NULL,
                               (WindowPtr)-1L );

    SetPort( My_Window );
    // HideWindow( My_Window );
    ShowWindow( My_Window );

    Main_Event_Loop();
}
```

Again select **Make** from the Project menu to create a new version of `TestApp3`. Now, when you drag the `PICTchooser` icon onto the `TestApp3` icon, everything will work as intended.

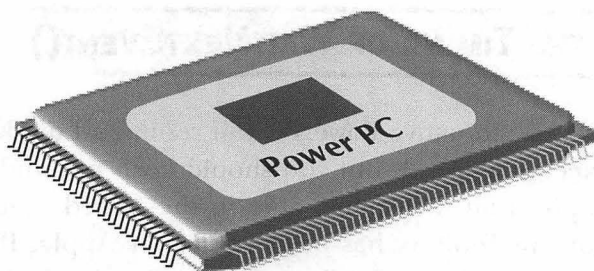
CHAPTER SUMMARY

If you're going to create a number of import libraries that will be used as plug-in tools for an application, you might consider giving the libraries their own icon that visually relates each library to the application that will use it. To do this, add a 'BNDL' resource to the resource file for the application project.

An import library can have three special routines. The initialization routine is called automatically when the library is loaded, and holds one-time-only code, such as memory allocation calls. The termination routine is invoked automatically when the library is unloaded. If memory was allocated in the initialization routine, it can be cleaned up here. The last special routine, the main routine, must be invoked explicitly by another fragment. Typically an application will make this call from within its own main event loop. The library's main routine often handles repetitive

tasks such as updating whatever was created during by the library's initialization routine.

Apple events can be added to any program that is to run under System 7. Apple events are especially useful for PowerPC applications that make use of import libraries, however. Giving your application the ability to handle an Open Document Apple event means that a user can drag an import library icon onto an application icon in order to launch that application and execute the import library code. You'll use Apple Event Manager functions to allow your program to support Apple events.



CHAPTER 13

OPTIMIZING POWERPC CODE

Earlier chapters showed you how to use universal procedure pointers and other programming techniques to port 680x0 code to PowerPC code. You've also seen how to eliminate PowerPC code redundancy by using import libraries. So, is your journey to PowerPC coding complete? Perhaps—or perhaps not. While your code may seem to run fine on a PowerPC, there's a chance that it could run even quicker. This chapter discusses a few of the techniques and tricks that will allow your PowerPC code to really shine.

IMPROVING THE TIMING OF WaitNextEvent()

Porting your code to native PowerPC will result in a marked improvement in execution speed. But you should keep in mind that even a fully ported application will spend time in the Mixed Mode Manager. Why? Not all of the Toolbox has been ported by Apple. If a PowerPC application makes a call to a Toolbox routine that hasn't been ported, that application must switch modes.

The execution of a 680x0 Toolbox instruction on a PowerPC doesn't involve just a couple of instructions. A mode switch involves the movement of parameters between the stack, the emulated 680x0 registers, and the PowerPC registers. The end result? The execution of a single nonported Toolbox function requires an average of 500 PowerPC instructions!

If your program makes only occasional calls to 680x0 Toolbox instructions, the amount of Mixed Mode time may very well be negligible and unnoticeable to your program's users. But if your application makes repeated calls to a 680x0 Toolbox trap, your program's performance will suffer—perhaps noticeably so. While it will be impossible to determine which Toolbox calls have been ported and which haven't (these categories will change over time), there is one particular Toolbox function you can be on the watch for—`WaitNextEvent()`.

Using WaitNextEvent() Outside the Main Event Loop

`WaitNextEvent()`, like all Event Manager routines—hasn't been ported to native PowerPC. But there are hundreds of other Toolbox routines that haven't been—so why single out this one call? Because many programs include extra calls to `WaitNextEvent()`—perhaps hundreds or thousands of such calls.

Any Mac program includes a call to `WaitNextEvent()` in the program's main event loop. But many other programs include an extra call to `WaitNextEvent()`—one that is made from within a time-consuming function. For instance, if a function has a loop that repeats thousands of

times, the programmer will typically insert a call to `WaitNextEvent()` within that loop. This extra call typically watches for a `keyDown` or `mouseDown` event. Should an event of that nature occur during the execution of the loop, it's assumed that the user wants to cancel the action being performed in the loop. Here's an example:

```
void Do_Time_Consuming_Stuff( void )
{
    long          i;
    EventRecord    evt;

    for ( i = 0; i < 10000; i++ )
    {
        MoveTo( 20, 20 );
        FillRect( &The_Rect, &qd.black );
        FillRect( &The_Rect, &qd.white );

        if ( WaitNextEvent( keyDownMask, &evt, 0L, nil ) )
            goto escape;
    }

    escape:
}
```

The `Do_Time_Consuming_Stuff()` routine simply executes a few `QuickDraw` commands—just something to kill a little time for the purpose of my testing out the use of `WaitNextEvent()`. The function constantly draws a globally defined rectangle, first in black, then in white. The result is a flickering rectangle that looks something like the one shown in Figure 13.1.

At each pass through the loop a call to `WaitNextEvent()` is made. I've used the `keyDownMask` to instruct `WaitNextEvent()` to watch only for a key stroke. Should the user press a key at any time during the execution of the loop, the loop will terminate. While `Do_Time_Consuming_Stuff()` doesn't perform the most useful or exciting task, it is representative of how `WaitNextEvent()` is often used outside of the main event loop. If you want to run the program yourself, look in the folder titled (13) *WaitNextEvent f*. In that one folder you'll find additional folders that hold all four of the short test programs that I'm about to describe.

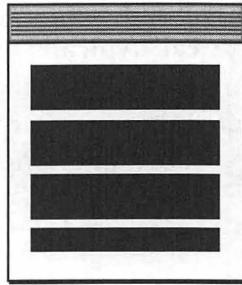


FIGURE 13.1 THE OUTPUT OF THE TEST PROGRAM.

Verifying the WaitNextEvent() Is Time Consuming

To see just how time consuming a call to `WaitNextEvent()` can be, I've written a couple of very short test programs. The first *doesn't* contain any calls to the `WaitNextEvent()` function. It loops 10,000 times, drawing the flickering rectangle at each pass through the loop:

```
void Do_Time_Consuming_Stuff( void )
{
    long    i;

    for ( i = 0; i < 10000; i++ )
    {
        MoveTo( 20, 20 );
        FillRect( &The_Rect, &qd.black );
        FillRect( &The_Rect, &qd.white );
    }
}
```

In order to gauge how long `Do_Time_Consuming_Stuff()` takes to execute, I added a few local variables and a couple of calls to `LMGetTicks()`. This function returns the number of ticks, or sixtieths of a second increments, that have passed since the system—the Mac—was booted. Figure 13.2 shows that when execution stops at the breakpoint at the bottom of the function, `seconds` has a value of about 7.8 seconds. I've taken the liberty of altering the screen dump so that the `seconds` variable appears in boldface—that's the variable I'm most interested in.

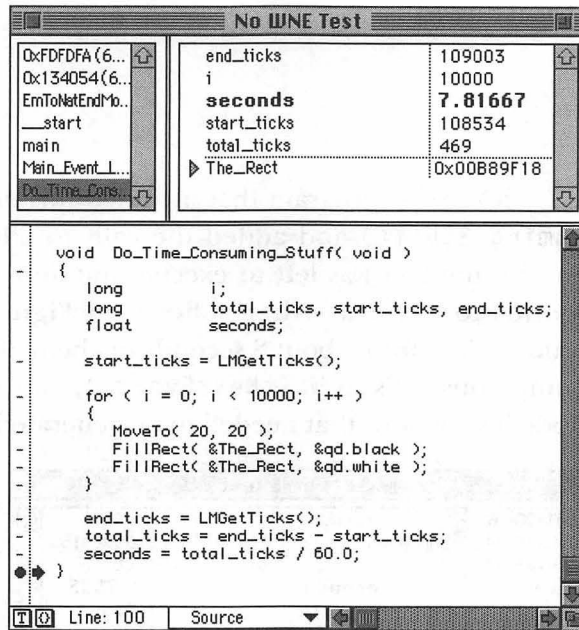


FIGURE 13.2 TIMING THE EXECUTION OF A FUNCTION THAT CONTAINS NO CALLS TO `WaitNextEvent()`.

Next, I've added a call to `WaitNextEvent()` inside the loop. Now, the user can interrupt the `Do_Time_Consuming_Stuff()` function—which takes about 8 seconds to execute—at any time by just pressing a key. Here's the revised function:

```
void Do_Time_Consuming_Stuff( void )
{
    long        i;
    EventRecord  evt;

    for ( i = 0; i < 10000; i++ )
    {
        MoveTo( 20, 20 );
        FillRect( &The_Rect, &qd.black );
        FillRect( &The_Rect, &qd.white );

        if ( WaitNextEvent( keyDownMask, &evt, 0L, nil )
```

Programming the PowerPC

```
        goto escape;
    }

    escape:
}
```

I set up a small CodeWarrior program that included this new version of `Do_Time_Consuming_Stuff()` and added the calls to `LMGetTicks()`. This time, when the function was left to execute uninterrupted, it took about 29.8 seconds to execute—that's shown in Figure 13.3. This increase in execution time from about 8 seconds to about 30 seconds was caused by the numerous calls to `WaitNextEvent()`, and by the subsequent Mixed Mode instructions that needed to be generated.

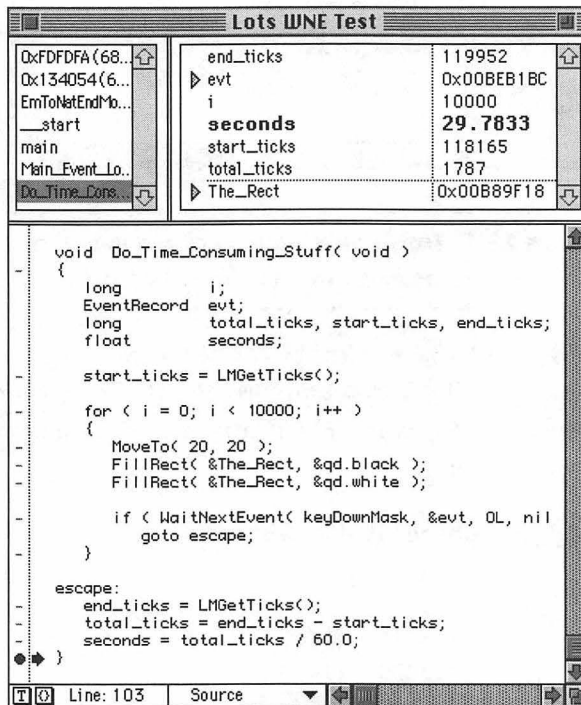


FIGURE 13.3 TIMING THE EXECUTION OF A FUNCTION THAT CONTAINS A CALL TO `WaitNextEvent()` IN EACH LOOP ITERATION.

A First Solution—Fewer Calls to WaitNextEvent()

One simple way to speed up a function's execution—while still giving the user the power to terminate the function—is to simply reduce the number of calls to `WaitNextEvent()`. There's no need to make this Toolbox call at every pass through a loop. Instead, keep a count of the number of times through the loop, and call `WaitNextEvent()` only every *x* passes through the loop. I chose a value of 10 for *x*, which means that `WaitNextEvent()` will be called only every tenth pass through the loop. Here's a look at the latest version of `Do_Time_Consuming_Stuff()`:

```
void Do_Time_Consuming_Stuff( void )
{
    long          i;
    EventRecord    evt;
    short          count = 0;

    for ( i = 0; i < 10000; i++ )
    {
        count++;

        MoveTo( 20, 20 );
        FillRect( &The_Rect, &qd.black );
        FillRect( &The_Rect, &qd.white );

        if ( count == 10 )
        {
            count = 0;
            if ( WaitNextEvent( keyDownMask, &evt, 0L, nil )
                goto escape;
        }
    }

    escape:
}
```

This greatly sped up the function execution, while preserving the function's ability to catch and respond to a `keyDown` event. Figure 13.4 shows that function execution time has been reduced from almost 30 seconds to just over 10 seconds.

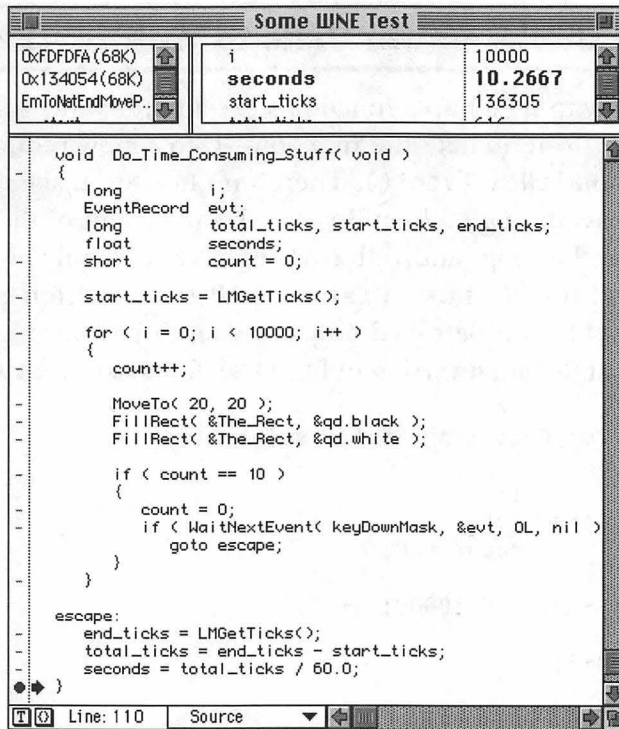


FIGURE 13.4 TIMING THE EXECUTION OF A FUNCTION THAT CONTAINS CALLS TO `WaitNextEvent()` EVERY TENTH LOOP ITERATION.

A Second Solution—Timing the Calls to `WaitNextEvent()`

A second solution to reducing `WaitNextEvent()` mixed mode time is to call the function at a specific interval of time. This solution fits more programming circumstances than the previous one. For instance, you may have a function that doesn't have an unvarying execution time. Some loop iterations or some section of the function may take longer to execute depending on the values of certain passed parameters or global variables. For such a case you'll want to check to see how much time has elapsed since the previous call to `WaitNextEvent()` was made. If a suffi-

cient amount of time has passed, call `WaitNextEvent()` again. Below is the final version of `Do_Time_Consuming_Stuff()`:

```
#define    TIME_BETWEEN_WNE    15

void Do_Time_Consuming_Stuff( void )
{
    long    i;
    EventRecord    evt;
    long    time_to_call_WNE = 0;

    for ( i = 0; i < 10000; i++ )
    {
        MoveTo( 20, 20 );
        FillRect( &The_Rect, &qd.black );
        FillRect( &The_Rect, &qd.white );

        if ( LMGetTicks() > time_to_call_WNE )
        {
            if ( WaitNextEvent( keyDownMask, &evt, 0L, nil )
                goto escape;
            time_to_call_WNE = LMGetTicks() + TIME_BETWEEN_WNE;
        }
    }

    escape:
}
```

This version of `Do_Time_Consuming_Stuff()` calls `WaitNextEvent()` only if one quarter of a second has passed since the last call. The constant `TIME_BETWEEN_WNE` establishes this time—15 sixtieths of a second is one quarter of a second. The `time_to_call_WNE` variable starts with a value of 0, which will of course be less than whatever value is returned by `LMGetTicks()`. So the first pass through the loop will always invoke `WaitNextEvent()`. After `WaitNextEvent()` is called, the value of `time_to_call_WNE` is changed to the current system tick count plus the 15 tick count buffer. That means `WaitNextEvent()` won't get invoked until at least a quarter of a second has elapsed. Figure 13.5 shows that using this timed method, `Do_Time_Consuming_Stuff()` takes only 8.5 seconds to execute—much better than the 30 seconds the function takes to execute when `WaitNextEvent()` is called at each pass through the

Programming the PowerPC

loop, and not too much slower than the 7.8 seconds the function takes to execute when no `WaitNextEvent()` calls are in the loop.



But wait! What about all those calls to `LMGetTicks()`? True, there's one at each pass through the loop. But this routine is ported to native PowerPC, so there's no mode switching involved. And you can see from Figure 13.5 that the timing of the loop doesn't suffer from the over 10,000 calls to `LMGetTicks()`.

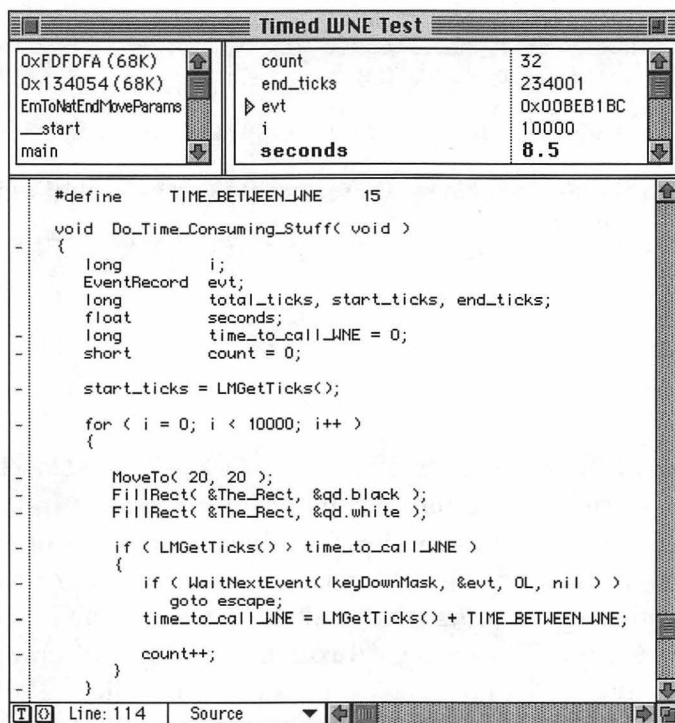


FIGURE 13.5 TIMING THE EXECUTION OF A FUNCTION THAT CONTAINS TIMED CALLS TO `WaitNextEvent()`.

MISCELLANEOUS PERFORMANCE ENHANCEMENTS

Limiting calls to `WaitNextEvent()` is the simplest single step you can take to improve the performance of your PowerPC application. There are, however, a few other tips you'll want to consider.

Align Data Structures

As described in Chapter 10, make sure that your compiler is set to PowerPC structure alignment whenever possible. The Power Mac can access struct members from a struct that has been appropriately padded much more quickly than it can access members of a struct that was specifically aligned for a 680x0-based Macintosh.

You'll only need to use `#pragma options align = mac68K` when you know your application will be transferring data between a 680x0-base Macintosh, or if it will be running on a network with both Power Macs and 680x0-based Macs.

Move Floating-Point Parameters to the End of the List

One subtle but sometimes noticeable change you can make to your code is to move all floating-point parameters to the end of the parameter list for application-defined functions. If you have a routine with the following parameters:

```
void My_Function( double d_1, double d_2, double d_3,
                  double_4, int i_1, int i_2, int, i_3 )
```

Change the ordering of the parameters to this:

```
void My_Function( int i_1, int i_2, int, i_3,
                  double d_1, double d_2, double d_3,
                  double_4 )
```

Programming the PowerPC

The reasoning behind this switch? You'll reduce memory accesses. The PowerPC contains eight general-purpose registers that are reserved for the first eight function parameter words. These eight registers will always be used first—no matter what type of parameters are first encountered. Once these eight words are filled, the PowerPC will use either the stack or some of its 13 floating-point registers to hold the remaining parameters. If the parameters aren't floating-point values, they'll be written to the stack—and that means memory access. If the parameters are floating-points, they'll stay in the PowerPC's floating-point registers.

In my first example, repeated below, the eight words of the general purpose registers will quickly be filled by the double parameters—each double occupies 8 bytes, or two PowerPC words. That means that the last three parameters, the integers, will all be written to the stack.

```
void My_Function( double d_1, double d_2, double d_3,
                  double_4, int i_1, int i_2, int, i_3 )
```

Now consider what happens when the ordering of the parameters is changed—as in the version of `My_Function()` that's shown below. Here, the first three parameters—the integers—would go into three of the PowerPC's general purpose registers. Next, the four double parameters would go into four of the 13 PowerPC floating-point registers. A simple rearranging of the parameters results in the elimination of memory access.

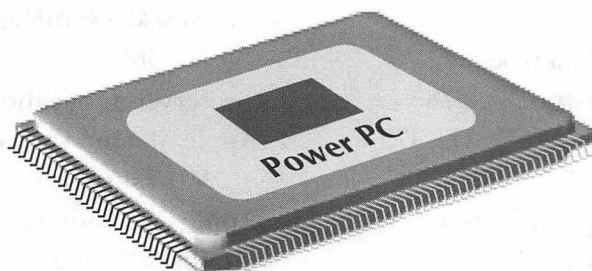
```
void My_Function( int i_1, int i_2, int, i_3,
                  double d_1, double d_2, double d_3,
                  double_4 )
```

CHAPTER SUMMARY

Using the techniques provided in the previous chapters of this book will allow you to port any existing 680x0 application to native PowerPC code. There are a few easily overlooked tips you'll want to follow, however, to push the most performance out of your Power Mac.

The single most important change you can make to your source code is to limit the number of calls that are made to `WaitNextEvent()`. The Event Manager hasn't been ported to native PowerPC code, so you'll want to institute some sort of timing code to limit extra calls to this routine.

You can look over the parameter list of each of your program's application-defined functions to verify that floating-point parameters (`float`, `double`, and `long double`), all appear at the end of the lists. That makes it easier for the PowerPC to work with function calls. Finally, make sure that your PowerPC-only applications don't have `struct` alignment set to 680x0 alignment. The PowerPC can access `struct` members much more quickly if those members fall on boundaries set for the PowerPC.



INDEX

Symbols

__powerc macro, 242-248
__POWERPC macro, 241-248
32-bit clean software, 49, 248-249
680x0 microprocessor chip, *see*
 Motorola microprocessors
68LC040 Emulator, *see* Emulator
 software
90\10 rule, 47

A

A5 world, 82
accessor functions, 252-253
AEDesc data type, 372
AEDescList(), 372-373
AEGGetKeyDesc(), 371-372
AEGGetNthPtr(), 372-373
AEInstallEventHandler(), 365-
 367, 369
AEProcessAppleEvent(), 363-365,
 369
Alert(), 136, 286
alerts, PowerPC-only warning,
 193-199
align pragma options, 261-263,
 389
ANSI C compliance, 230-237
Apple Desktop Bus (ADB), 7
Apple event handlers, 364-367,
 370-375

Apple Events

- AEDesc data type, 372
- AEDescList(), 372-373
- AEGetKeyDesc(), 371-372
- AEGetNthPtr(), 372-373
- AEInstallEventHandler(), 365-367, 369
- AEProcessAppleEvent(), 363-365, 369
- applications and, 367-375
- client application, 362
- core events, 362
- defined, 362
- DescType data type, 372
- dispatch table, 366
- event handlers, 364-367, 370-375
- FSSpec data type, 373
- installing event handlers, 369-370
- kAEOpenApplication constant, 365
- kAEOpenDocuments constant, 365
- kAEPrintDocuments constant, 365
- kAEQuitApplication constant, 365
- kCoreEventClass constant, 365
- kHighLevelEvent constant, 363

- NewAEEEventHandlerProc(), 366

- server application, 362

- Apple Macintosh Programmer's Workshop (MPW), 230

- Apple RISC Software Developers Kit (SDK), 229

- AppleScript, 105, 106-109

- application partition, 249

- application code fragment, 73

- architecture, *see* runtime environment

- arithmetic logic unit (ALU), 38

B

- bit ordering, 31-32

- BNDL resource, 332-339

- branch instructions, 32-36

- branch prediction, 35-36

- Branch Processing Unit (BPU), 32-36, 38

- business users, 8

C

- cache memory

- coherency, 39

- data, 39-41

- defined, 39

- instruction, 41-43

- snooping logic, 39-40

- CautionAlert(), 136

- CDKdemoPPC, 111-131

cfrg resource, 99, 119, 121, 173-175, 202-203, 308-309
CISC technology
 condition code register (CCR), 14, 16-17, 21
 defined, 2
 evolution of, 14
 instruction execution, 14-17
 instruction timing, 17-19
 limiting factors of, 16-17
 status register, 14
client application, 362
CloseConnection(), 291, 306
CloseResFile(), 286
Code Fragment Manager (CFM)
 defined, 74
 loading fragments, 75-78
code fragments
 containers for, 275-277
 creation of, 274
 defined, 65, 73
 symbols in, 75, 79-80
code resource fragment, 74
CODE resources, 172, 173-175, 200-201, 309
coherency, 39
CompanyInfo, 292-300
compatibility issues, 9-11
compilers
 PowerPC, 87-88
 see also Symantec CDK compiler
 see also Metrowerks

CodeWarrior compiler
complex instruction-set computer, *see* CISC technology
condition code register (CCR), 14, 16-17, 21
condition register (CR), 21
conditional compilation directives, 145, 238-248
containers, code fragment, 275-277
ControlActionUPP, 148-149
core events, 362
cross-mode calls, 58-61

D

data alignment
 680x0, 256-258
 compatibility, 261-263
 lasting effects of, 265
 manual, 267-271
 padding, 256-260
 PowerPC, 258-260
 problems with, 260-261
 structures/variables, 265
 testing, 263-266
data fork erasing, 202-206
data section, 76-77
debugging, 264
DFerase utility, 204-206
dispatch table, 366
DisposRoutineDescriptor(), 143
DOS/Mac compatibility, 8-9
double data format, 211, 212-213

double-double data format, 211-213

double_t data type, 216-217

DrawText(), 47

dynamically linked libraries, *see*
import libraries

E

Emulator software

cross-mode calls, 58-61

defined, 49-50, 225

mode switches, 50-55, 58

engineers, 8

EraseRgn(), 47

error -192 ID, 169-170, 194

error checking, 344-345

Ethernet, 7

event handlers, 364-367, 370-375

EventRecord data type, 347

example programs

CDKdemoPPC, 111-131

CompanyInfo, 292-300

fat binary stripping, 200-208

fat binary, 176-193

graceful exit, 193-199

MWdemoPPC, 90-104

PICTchooser, 339-349

TestApp1, 301-312

TestApp2, 313-324, 349-358

TestApp3, 367-375

UPPdemo1, 152-156

UPPdemo2, 157-164

exiting gracefully, 193-199

expression evaluation, 218-223

Extended Common Object File
Format, 277

extended data type, 211, 216-217

extension fragment, 74

F

fat binary applications

creating with CodeWarrior,
176-184

creating with Symantec, 184-
193

defined, 145, 167, 170

forks of, 173-176

stripping to PowerPC-only,
167, 200-209

filter functions, 134-136, 140-143,
146-148, 154-156

float.h header file, 228

floating-point registers (FPR), 39

Floating-Point Unit (FPU), 37-39

forks, resource and data, 171-176

fp.h header file, 213, 216-217

FragLoad.h header file, 283, 304

fragments, *see* code fragments

FSMakeFSSpec(), 288, 305

FSpOpenDF(), 341

FSpOpenResFile(), 285

FSRead(), 342

FSSpec data type, 285, 288, 304-
305, 373

function

calls, 33-36

declarations, 233-236
prototypes, 236-237
return types, 233-236

G

general purpose registers
 (GPR), 38
Gestalt(), 57
gestaltProcessorType selector, 57
GetDiskFragment(), 289-291, 305-
 306, 312
GetEOF(), 341
GetFontInfo(), 47
GetNewCWindow(), 353
GetSharedLibrary(), 326-328
global variables, 76-77, 82
glue code, 60-61
GrayRgn, 251-253

H

HideWindow(), 375
HiWord(), 320

I

icons, import libraries, 332-339,
 348-349
import libraries
 advantages of, 69-73, 286-287,
 315-316
 CloseConnection(), 306
 connections, 278-279
 creating, 291-300

 Creator, 295
 data fork size, 300
 defined, 65, 68, 277
 entry points, 279, 296
 export symbols, 278
 GetDiskFragment(), 289-291,
 305-306, 312
 GetSharedLibrary(), 326-328
 icons for, 332-339, 348-349
 import symbols, 278
 InitBlockPtr data type, 283,
 343
 initialization routine, 279-280,
 297-298, 342-345
 loading via menus, 312-313
 loading, 288-291
 main routine, 279, 281, 346-
 347, 353-358, 359-360
 plug-in tools, 315-316
 qd globals and, 358-360
 resource access, 280, 284-286
 shlb type, 296, 324-328
 special routines, 279
 symbols, 278
 termination routine, 279, 281,
 345-346
 testing, 300-312
 Type, 295
 unloading, 291
 uses for, 316
InitBlockPtr data type, 283, 343
initialization routine, 279-280,
 297-298, 342-345

Programming the PowerPC

Insignia Solutions, 8
installing event handlers, 369-370
instruction set architecture, 58,
 139
INT_MAX constant, 229
integer exception register (XER),
 38
Integer Unit (IU), 37-38
InterfaceLib, 93

K

kAEOpenApplication constant,
 365
kAEOpenDocuments constant,
 365
kAEPrintDocuments constant,
 365
kAEQuitApplication constant,
 365
kCoreEventClass constant, 365
kHighLevelEvent constant, 363
KillPicture(), 345

L

latency, 25
libraries, 213-214
limits.h header file, 228, 231
Line(), 47
linked libraries
 creating, 68
 defined, 68
 MacTraps, 68

LMGetGrayRgn(), 252
LMGetTicks(), 387-388
LoMem.h header file, 253
low-memory system globals, 249-
 253
LowMem.h header file, 252
LoWord(), 320

M

Mac/DOS compatibility, 8-9
Mac/Windows compatibility, 8-9
mac68k pragma, 261-263, 389
MacHeadersPPC, 96-97, 177, 248
Macintosh Programmer's
 Workshop (MPW), 230
MacTraps, 68
main routine, 279, 281, 346-347,
 353-358, 359-360
MathLib, 93, 213
MAXINT constant, 229, 230-232
memory address bit size, 249
Memory control panel, 249
Memory Management Unit
 (MMU), 56
Memory Manager, 47
Metrowerks CodeWarrior compiler
 adding files to, 92-96
 (application building, 103-104
 cfrg resource, 99, 308-309
 defined, 87, 88-89
 example program, 90-104
 fat application building, 176-
 184

fat binaries and, 104
import libraries, 292-300, 347-349
InterfaceLib, 93
libraries and, 96-98
MacHeadersPPC, 96-97, 177, 248
MathLib, 93
MWCRuntime.Lib, 93
preferences, 96-98
prefix file, 96-97
project creation, 90-92
resource files and, 98-100
SIZE resource, 99, 308-309
xSYM file, 310
minimum evaluation format, 219-223
Mixed Mode Manager
 cross-mode calls, 58-61
 defined, 49-50, 58, 133
 mode switches, 50-55, 58
ModalDialog(), 134-136, 140, 142, 152-156
ModalFilterUPP, 141
mode switch, 52
Motorola microprocessors, 56
multimedia authors, 8
multiply-add array, 39
MWCRuntime.Lib, 93
MWdemoPPC, 90-104

N

nanokernel system software, 50

native software, defined, 47
NewAEEEventHandlerProc(), 366
NewControlActionProc(), 148-149
NewHandle(), 341-342
NewModalFilterProc(), 141
nil, in place of UPP, 149
NoteAlert(), 136
numeric environments
 defined, 209
 see also SANE
 see also PowerPC Numerics

O

OpenDeskAcc(), 321
optimizing PowerPC code
 data structure alignment, 256-271, 389
 LMGetTicks(), 387-388
 parameter ordering, 389-390
 ported traps, 380
 WaitNextEvent(), 380-388

P

partitions, memory, 249
PEF file format, 277
PicHandle data type, 342
PICT files, opening, 339-342
PICTchooser, 339-349
pictures, displaying, 339-342
pipelining, 25
plug-in tools, *see* import libraries
ported system software, 46-47

porting code

- 32-bit clean, 248-249
- ANSI C compliance, 230-237
- assembly language, 229
- comp data type, 218
- conditional compilation, 238-248
- declarations, function, 233-236
- extended data type, 216-217
- int data type, 230-232
- long data type, 232
- low-memory system global variables, 249-253
- prototypes, function 236-237
- qd global variable, 239-241
- QuickDraw globals, 239-241
- return types, function, 233-236
- short data type, 232
- single source code file, 237-238
- thePort global variable, 239-241
- universal header files, 226
- see also* universal procedure pointers

Power Macintosh

- compatibility issues, 9-11
- developer support, 10-11
- multimedia versions, 6
- naming of, 5
- original release, 5-9

sales of, 1

- System software version, 9, 49

powerc macro, 242-248

POWERPC macro, 241-248

PowerPC Numerics

- 680x0 and, 210
- comp data type, 218
- defined, 209
- double data format, 211, 212-213
- double-double data format, 211-213
- double_t data type, 216-217
- expression evaluation, 218-223
- extended data type, 211, 216-217
- fp.h header file, 213, 216-217
- libraries, 213-214
- MathLib, 213
- minimum evaluation format, 219-223
- PowerPC and, 210
- scalb(), 217
- single data format, 211, 212-213

pragma options, 261-263, 389

Preferred Executable Format, 277

prefetching, 31

prefix files, 96

printf(), 263

procedure pointers

- defined, 62, 136

errors compiling, 144
ModalDialog(), 134-136, 142
UPPs and, 136-140

Process Manager, 173-174
ProcPtrs, *see* procedure pointers
program segmentation, 67,
76-77
prototypes, function 236-237
publishers, 8

Q

qd global variable, 124, 188, 239-
241, 358-360
QuickDraw, 47

R

reduced instruction-set computer,
see RISC technology
reset pragma, 261-263, 389
resource not found error, 169-
170, 194
return types, function, 233-236
RISC instructions
Branch Processing Unit and,
32-36
building, 3-4, 22
dependencies, 22
execution, 21-24
fetching, 30-36
latency, 25
pipelining, 25
prefetching, 31

queue, 31, 32-36
rearrangement, 22
scheduling, 22-24
stalls, 24
throughput, 25
timing, 25-27

RISC Software Developers Kit
(SDK), 229

RISC technology
basic operation of, 3
benefits of, 4
condition register (CR), 21
defined, 3
evolution of, 20-21
latency, 25
pipelining, 25
stalls, 24
throughput, 25
see also RISC instructions

routine descriptors
creating, 141, 149
defined, 59, 62-63
described, 139
NewModalFilterProc(), 141

runtime environment
defined, 66
overhaul of, 66-67

S

SANE
comp data type, 218
defined, 209
expression evaluation, 218

- extended data type, 211, 216-217
- libraries, 213
- PowerPC and, 210
- SANE.h header file, 213, 216-217
- SANE.lib, 213
- scalb(), 217
 - see also* PowerPC Numerics
- scalb(), 217
- Segment Manager, 172
- segmentation, 67, 76-77
- SetDItem(), 150-152
- SetFPos(), 341
- SFTypeList data type, 340
- shared libraries, *see* import libraries
- Shared Library Manager, 78
- ShowWindow(), 375
- SHRT_MAX constant, 231-232
- SIZE resource, 99, 119-120, 173, 308-309
- snooping logic, 39-40
- SoftWindows emulation software, 8
- special purpose register (SPR), 39
- special routines, 279
- stalls, 24
- Standard Apple Numerics Environment, *see* SANE
- StandardFileReplay data type, 340
- StandardGetFile(), 136, 340
- status register, 14
- stereo input/output, 7
- StopAlert(), 136
- superscaling, 36-39
- Symantec CDK compiler
 - adding files, 115-117
 - AppleScript and, 105, 106-109
 - application building, 121, 126-131
 - build errors, 129-130
 - cfrg resource, 119, 121
 - defined, 87, 105
 - example program, 111-131
 - fat application building, 184-193
 - int, size of, 230-231
 - memory partitions, 119-120
 - project creation, 112
 - project folder, 109-112
 - qd global variable, 124, 188
 - resource files and, 112-115
 - SIZE resource, 119-120
 - ToolServer, 105
 - translators, 244-247
- symbols, 278
- SysEqu.h header file, 252
- System 7.1.2, 9, 49
- system global variables, 249-253
- system heap, 249
- system partition, 249
- system software
 - porting of, 46-49
 - PowerPC version, 9, 49

T

Table of Contents (TOC)
 defined, 79-80
 pointers and, 81-84
 TVectors and, 79-80
termination routine, 279, 281,
 345-346
TestApp1, 301-312
TestApp2, 313-324, 349-358
TestApp3, 367-375
thePort global variable, 239-241
throughput, 25
ToolServer, 105
TrackControl(), 136, 147-149
transition vectors
 defined, 78-79
 pointers and, 78-79, 83-84,
 136
 Table of Contents and, 79
translators, 244-247
TVectors, *see* transition vectors

U

universal header files, 226
universal procedure pointers
 (UPP)
 conditional use of, 145-146
 ControlActionUPP, 148-149
 defined, 138-139
 disposing, 143
 DisposRoutineDescriptor(),
 143

 errors compiling, 144, 150
 local/global usage, 254-255
 ModalDialog(), 140, 152-156
 ModalFilterUPP, 141
 NewControlActionProc(),
 148-149
 NewModalFilterProc(), 141
 nil, in place of UPP, 149
 SetDItem(), 150-152
 user items and, 150-152, 157-
 164
 UserItemUPP, 150-152, 163
UniversalProcPtr, *see* universal
 procedure pointers
UnloadSeg(), 76-77
UPPdemo1, 152-156
UPPdemo2, 157-164
user items, 150-152, 157-164
UseResFile(), 285
UserItemUPP, 150-152, 163
USERROUTINEDESCRIPTORS
 macro, 238-239

V

Values.h header file, 228, 231
variables, missing, 264
vers resource, 292-293
Virtual Memory Manager, 49

W

WaitNextEvent(), 380-388
Windows/Mac compatibility, 8-9

X

XCOFF file format, 277

xSYM file, 310

ABOUT THIS DISK

The one 1.4 M disk contains a single folder named PowerPC Programming *f*. Within this folder are three more folders. The first contains a simple utility program that you'll use in Chapter 8. The other two folders holds source code files and project files for each of the examples presented in the this text. One folder holds Metrowerks CodeWarrior projects, the other holds Symantec Cross-Development Kit (CDK) projects. If you have either of these compilers, this disk provides you with everything you need to get started.

This disk is a Macintosh 1.4M high-density disk. All newer model Macintosh computers come with the SuperDrive—a 1.4 M high-density floppy drive. If you have an older Macintosh with an 800 K double-density floppy drive, you won't be able to use this disk. You can, however, if you find a friend or coworker who has a SuperDrive. That person can copy the folders to two 800 K disks for you. The files on this 1.4M disk are not compressed or archived—just copy them to your hard drive and use them “as is.”



A Division of MIS:Press, Inc.
A Subsidiary of Henry Holt and Co., Inc.

Programming the Power PC

Dan Parks Sydow

ISBN 1-55851-400-7
Copyright ©1994 M&T Books
Format: Macintosh

M&T Books

115 West 18th Street New York, NY 10011

Programming The PowerPC

Programming Native Applications for the New Power Macintosh

Programming the PowerPC is a complete introduction to the exciting new world of RISC computing brought to reality by the new line of Power Macintoshes introduced by Apple. Far from a simple overview of RISC architecture, this book leaves history of RISC technology behind. Instead, you'll jump right into practical advice and workable coding techniques for maximizing the capabilities of the new Power Macintosh computers. You'll get hands-on instruction for producing "native" code and applications that will fly on the new Macs.

From the basics of RISC to advice on porting existing applications, everything you need to be on the cutting edge is here—waiting for you to jump in. You'll get all the details of how to write programs that take full advantage of the capabilities of the PowerPC. You'll learn about the runtime environment of the PowerPC system software, about the 68LC040 emulator, and how to program effectively for both.

DAN PARKS SYDOW is a professional writer and programmer in the computer field. He is the author of *Macintosh Programming Techniques*, and has written numerous educational Macintosh programs.

Topics Include:

- The details of PowerPC architecture
- 68LC040 emulation
- How to create native applications
- How to create "fat binaries"
- Compiling Power Mac code with both MetroWerks CodeWarrior and Symantec C++
- The Power System Software
- The implications of RISC and CISC technology

Cover art © Garry Gay, The Image Bank
Cover design by Gary Szczecina



Level

Intermediate-Advanced

Programming

Power Mac

ISBN 1-55851-400-7



9 781558 514003

US \$39.95
CAN \$54.00

90000>

