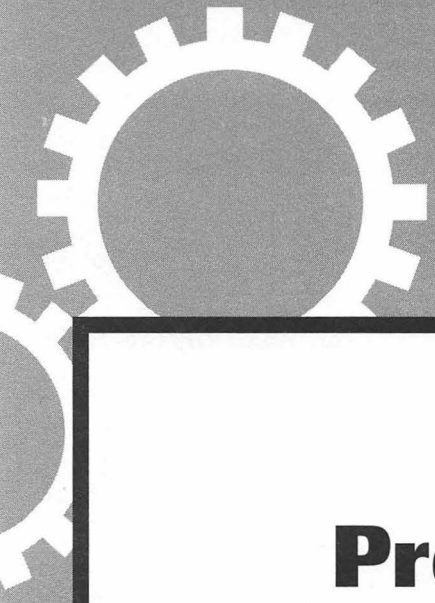
PowerPC™ Programmer's Toolkit

The definitive resource on the latest PowerPC techniques

CD-ROM
includes a special version of
Metrowerks CodeWarrior™ Lite

Tom Thompson



PowerPC Programmer's Toolkit

PowerPC Programmer's Toolkit

Tom Thompson

PowerPC Programmer's Toolkit

Library of Congress Catalog Number: 95-80297

ISBN: 1-56830-241-x

98 97 96 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit number is the number of the book's printing. For example, a printing code of 96-1 shows that the first printing of the book occurred in 1996.

Trademark Acknowledgments

The Hayden Books Team

Publisher:	Lyn Blake
Publishing Manager:	Laurie Petrycki
Development Editor:	Kezia Endsley
Copy and Production Editors:	Bront Davis, Lisa Wilson
Technical Reviewers:	Jim Trudeau, Metrowerks, Inc. Alan Lillich, Apple Computer, Inc. Mark Anderson, Metrowerks, Inc.
Cover Designer:	Karen Ruggles
Interior Designer:	Sandra Stevenson
Production Analysts:	Mary Beth Wakefield
Production Team Supervisor:	Laurie Casey
Production Team:	Heather Butler, Angela Calvert, Kim Cofer, Dan Caparo, David Garratt, Aleata Howard, Michelle Lee, Erika Millen, Beth Rago, Erich Richter, Karen Walsh
Indexer:	Brad Herriman

Dedication

To my wife, Brenda Jean, and my children, John and Evelyn.

About the Author

Tom Thompson has a BSEE degree and bought his first 128K Macintosh in early 1984. He is a Senior Tech Editor at Large for *BYTE* magazine and has been covering the Mac for over ten years. He is an Associate Apple Developer, and has substantial programming experience, including authoring several shareware utilities. He has also researched and written numerous articles on programming and hardware technology.

Acknowledgments

A book is a lot like a programming project. It involves a lot of people working in concert to achieve the final outcome—all on budget and on schedule. While publishing doesn't normally involve writing code and using debuggers, in some ways it is more work because you have to explain things in a way that makes the most sense to the most people. People are pretty imprecise beings, unlike computers. Of course, this is a programming book where you do have to write code, use debuggers, as well as try to make things sensible. It can be done, but not without the capable assistance of many good people whom I'd like to thank.

To Karen Whitehouse at Hayden Books for her support for the second edition of this book. Thanks to Lisa Wilson and Bront Davis for making my prose sensible. An award to my editor Kezia Endsley, for tirelessly shepherding this book into reality.

To Greg Galanos, Jean Bélanger, Dan Podwall, John McEnerney, Berardino Baratta, and the rest of the Metrowerks gang for providing timely support and updates to their excellent CodeWarrior software during the course of writing this book. To Jordan Mattson at Apple for his support and access to PowerPC material.

To Eric Shapiro of Rock Ridge Enterprises for his valuable code contributions and suggestions. Eric taught me everything about 68K trap patching, and did it again for PowerPC trap patching. A lot of his code appears in the FlipDepth Extension shown in Chapter 6, and he made many recommendations that improved the SwitchBank application. Without his efforts and timely support, Chapter 6 would not have been possible.

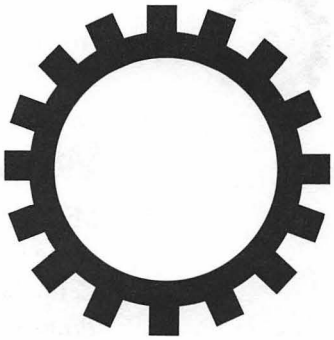
Thanks to Roger Goode for the improved graphics. Thanks to my tech reviewer, Jim Trudeau, for his perceptive remarks. Special thanks to Richard Hooker and Marvin Denman for checking over the cache descriptions. Special thanks to Randy Thelen for his insights into the Power Mac run-time architecture, which helped shape Chapter 4.

Thanks to Steve Jasik for providing a copy of The Debugger, software that will really make a difference in debugging PowerPC programs.

Last but not least, I would like to thank my wife Brenda for her patience during the long nights I spent toiling away on this new edition.

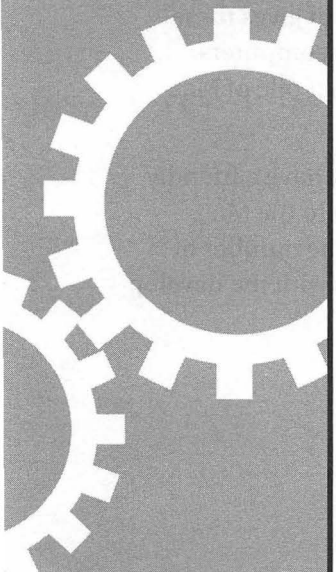
Overview

Contents



Introduction

This book is a road trip. In it, you'll find information on the PowerPC chip, RISC technology, and a C development environment by Metrowerks called Metrowerks CodeWarrior. You'll find an assortment of programming hints and tips and insights into how the Mac works, and you'll discover what new features—and pitfalls—await on the PowerPC chip. Most important, while I'll supply plenty of programming examples, I'll also explain how the Power Mac works. I firmly believe that if you understand how something works, you're in a better position to use it (or in the case of a personal computer, program it).



What You'll Need

My basic assumption is that you know how to use a Macintosh and have some knowledge of the C programming language. If you're not familiar with C, the best reference on this language is Kernighan and Ritchie's *The C Programming Language, Second Edition*, published by Prentice Hall. You also should have Apple's reference works on the Mac Toolbox, *Inside Macintosh*. I also assume you're familiar with using the CodeWarrior development tools. You'll have a demonstration version of Metrowerks CodeWarrior called CodeWarrior Lite on the CD-ROM accompanying this book. Check out its documentation files if CodeWarrior is new to you. If you don't have a Power Mac (yet), that's OK. Much of the material in here works with existing Macs as well, which is perhaps the real beauty of the PowerPC design.

I have structured this book so that it offers material useful to both novices and experienced Mac programmers. The novice should start at the beginning, but more experienced programmers should feel free to browse about and find a subject of interest. Consult the brief summaries at the beginning of each chapter to determine if the material is of interest to you. The following brief road map will help you decide your course.

The Road Map

Chapter 1 covers the Power Macs themselves, including the latest systems such as the Power Mac 9500, 8500, and 7500 that incorporate the industry-standard PCI bus. It also provides a brief peek at the PowerPC family of processors—the PowerPC 601, 603, 603e, 604, and 604e—that gives these systems their great horsepower. It also discusses how these computers manage to run existing Mac software, thereby preserving that pile of Mac software you've accumulated over the years.

Chapter 2 helps you write your first real C program. It won't have a friendly Mac interface, but it will perform a useful job. If you're new to the Mac, bypassing the user interface details for the moment limits the number of unknowns you have to deal with while you gain confidence with the development tools.

In Chapter 3, you'll tackle some of those user interface details dodged in Chapter 2. You'll add a friendly interface and discover the forked nature of Mac files. If you don't know what a Mac file's data fork and resource fork are, don't worry. This chapter will explain them to you. You'll also learn about resources (which, not surprisingly, reside in resource forks) and how to edit them for use in your program.

Chapter 4 is a rest stop on our journey. You will have reached a point where you must lay aside your tools for the moment and gain some insights into the Power Mac's new system architecture. I'll explain how Apple managed the feat where one set of source code can support two different processors. I'll go on to describe how the underpinnings of the Power Mac, as much as it resembles the 680x0 Mac on the surface, are fundamentally a different operating system. I'll explain what code fragments are, and what they mean to future application design. In addition, I'll describe Apple's Mixed Mode Manager, the part of the operating system that manages to keep two wildly different sets of processor code—the 680x0 and the PPC—operating in harmony. It will be of general interest to most readers, and required reading for those writing special programs and extensions. Finally, I'll explain how both 680x0 and PowerPC code can be embedded in a single application file—that fat binary mentioned earlier—so that such an application is capable of running on either Mac. You'll use some of these details later when you explore certain Power Mac-specific features.

Chapter 5 is where you put into practice the information you learned in Chapter 4. Most of this material will be of interest to advanced programmers. You'll write an application that controls the Mac's File Sharing software. This will require writing a function that works with the Mixed Mode Manager to enable a switch between 68K and PowerPC code. I'll also show how to make this application a fat binary, capable of running on both 680x0 and Power Macs. Next, you'll write an Extension that changes a Power Mac's screen depth. You'll see how to access code fragments. It also demonstrates how to patch the operating system, both for a 680x0 Mac and Power Mac.

In Chapter 6, it's time to focus on how to fix a program that misbehaves. Information on the types of debuggers, and debugging tools can be found here. A look at CodeWarrior's high-level debugger is provided. Tips on debugging and defensive coding are discussed.

Chapter 7 delves into processor-specific details and how they affect your application's operation. These details will help you decide strategies to get the best performance out of your code.

For those who want a better understanding of the processors, Appendix A provides a look at the PowerPC family.

Appendix B consolidates information on how to port an existing Mac application's C code to the Power Mac. It will be of interest to advanced programmers who just want to dive in and start retooling their programs immediately.

Appendix C provides the complete source listings for the programs discussed in this book.

Appendix D tells you how you can locate more CodeWarrior and Power Mac programming information.

The Metrowerks CodeWarrior Lite on the CD

The *PowerPC Programmer's Toolkit CD* contains the CodeWarrior Lite version 1.3 integrated development environment (IDE). CodeWarrior Lite prevents you from creating new source code files or projects. It also prevents you from adding new files to an existing project. Other than these restrictions, CodeWarrior Lite has the same capabilities as a full-fledged version of Metrowerks CodeWarrior 1.3 (aka CW8).

The text of this book was written using the full version of Metrowerks CodeWarrior. You'll have to use slightly different steps when using CodeWarrior Lite from the CD. The commands New, New Projects, and Add File... are not available. Because of these limitations, it can work only with the sample files provided on the CD.

So, if you are following along using CodeWarrior Lite, when the text tells you to use the New, New Project..., or the Add File... command, you should instead open the related project file and keep it open throughout the exercise. All the associated files will already be in the project, so you won't need the Add File... command. Then, you can follow the same procedures as if you were using the full version of CodeWarrior.

Note: We've provided all the code discussed in the book on the CD, so you don't have to retype it, unless you find it valuable to do so.

You also should note that Metrowerks cannot provide technical support for the Lite version. You can, however, get all the CodeWarrior information you could ever want and also meet other CodeWarriors. After you buy a full-up version, Metrowerks will be happy to provide full technical support.

Additional Notes

There probably are better ways to write some of the functions presented here and I welcome input from you. The purpose of my code, however, is to illustrate Power Mac features while being readable by an audience of C programmers with a wide range of experience. I also bias my code toward readability because, more often than not, six months later I usually have to modify the code for use in other projects.

While I've tried to produce error-free code, and I actually use some of these programs in my day-to-day work, it's possible that some of the code samples have bugs. Please send me bug reports via email or some other means. If you have access to AppleLink, my email address is T.THOMPSON, while on the Internet it is `tom_thompson@bix.com`. If you prefer a more conventional method, mail your comments and bug reports to me in care of Hayden Books.

Please note these signposts along the road as we travel.

Background Info

Magnifying Glass icons flag sections of the book where additional background information can be found. For those unfamiliar with a topic, this extra information promotes a better understanding of the material. Seasoned Mac programmers can skip these sections.

Important
Lightning Bolt icons signal important topics. These sections provide information necessary to understand the material in each chapter, or illustrate an essential point of the software or operating system. Even seasoned programmers might want to examine these sections for Power Mac-specific details.

Hazard
Bomb icons signal potential hazards. These sections supply crucial information required to keep your program from crashing and your Power Mac system intact. Do not skip these parts of the book.

Future Directions
Eight Ball icons indicate information that is applicable to the direction that Apple is taking the Macintosh platform, and operating system. You might, for example, find information on the Common Hardware Reference Platform (CHRP) here, or on Apple's next release of its operating system, code-named Copland.

User input text appears in a bold monospace font, as in:

```
Type MyFile and press Return.
```

Directives, routines, streams, and functions appear in a monospace font, as in:

```
Before we call Munge_File(), we fetch the stopwatch cursor icon using GetCursor().
```

Filenames appear in quotation marks, as in:

> For a complete source code listing, check the file "switchBank.c" on the CD-ROM.

The symbol ➥ has been used to represent program lines that have wrapped.

Well, enough preliminaries. Let's hit the road....

The Power Macs and a Brief History

History of the Power Macs

In early 1994, Apple changed the face of the personal computer industry—again. The company took a powerful processor technology previously available only in expensive workstations and offered it to small businesses and average users through affordable desktop computers. These low-cost computers, however, won't run those arcane workstation operating systems. Instead, they offer an interface renowned for its ease of use: the Macintosh operating system, or Mac OS. Put simply, Apple has introduced a new line of high-performance Macintosh computers, the Power Macs.

Because these Power Macs borrow heavily from the Macintosh design, a brief history of the Mac is in order.

The Early Mac

Just a decade ago, Apple introduced its newest personal computer during the 1984 Super Bowl. This famous commercial, titled "1984" and directed by Ridley Scott, depicted a bleak, gray, future dystopia where shaven-headed drones shuffled toward the ultimate video conference. A runner—hammer held high and wearing an Apple logo on her shirt—raced onto the scene, hotly pursued by the faceless thought police. The hammer was hurled at the conference screen, shattering it. The implication was that Apple's then-new Macintosh computer would save us from that same gloomy fate. The verdict is still out on whether the Mac accomplished that goal, but no one disputes its effect on how we deal with computers and information. Desktop publishing, digital image editing, color printing, and other applications were either invented on the Mac or driven by the demands of its users.

The original Macintosh (now termed "classic Mac" in Apple's technical literature) was a small beige box with a 7.83 MHz 68000 processor. It came equipped with a built-in 9-inch black-and-white monitor, 128 K of random-access memory (RAM), a single custom 3 1/2-inch Sony floppy drive, two serial ports, and 64 K of read-only memory (ROM). The classic Mac was a "closed system" because it offered no slots or easy expansion capabilities.

The Mac ROMs provided a large array of support routines that implemented the graphic user interface (GUI) and system services such as memory management and file I/O. These routines are known collectively as the Mac Toolbox. Because it's easier to use the Toolbox services than write code from scratch, the Mac always has encouraged a consistency in application design. Much of the Mac's "personality," or behavior, comes from these Toolbox routines.

Important
Because Toolbox routines are relied on heavily when writing a Mac application, expect to become familiar with them as you progress through this book. Keep *Inside Macintosh* nearby; those manuals provide important details on Toolbox routines. As you become comfortable with programming the Mac, you'll frequently consult them when writing new applications and adding features to existing applications.

Because well-behaved applications only access the system through the Toolbox interface, Apple has retained the option of significantly revising the hardware and software behind the interface without requiring modifications to existing applications. A new Mac, for example, might use a new stereo sound chip, but your application would still use the same sound generation routines and be able to play music or sound effects on it. That's because while the Toolbox sound routines still present the same interface to the programmer, the code underneath this interface layer converts your program's commands into a format the new hardware understands. This design eliminated many compatibility problems as Apple enhanced both the Toolbox routines and the hardware. Of course, not all compatibility problems were avoided, but Apple was able to limit them by using the Toolbox to define a "virtual machine."

Just as important, the Mac GUI helped enforce a consistency in the application's user interface, making Mac applications easier to use than those on other computers. After you mastered one application, you knew the basics of using other Mac applications as well. To be sure, there were application-specific features users had to learn (text formatting in a word processor, or how to use a pen tool in a drawing package), but they didn't start over each time with the basics. They could always count on finding file manipulation commands under an application's File menu, and locating the editing commands in the Edit menu.

Faster and Better

Over the years, Apple improved the original Mac and introduced new models. First the company added more memory and a SCSI port. With the Mac II, Apple used the faster 68020 processor and opened the computer's closed design by adding NuBus slots. It introduced newer Macs with faster processors and a larger array of features. These machines went by such arcane names as the Mac IIcx, IIci, IIsi, IIvi, and IIvx. Apple minimized the confusion temporarily by giving certain product lines unique group names. The Mac notebook computers were labeled PowerBooks. Numbers were tacked onto the end of the names to help identify the characteristics of each computer. Still, things got out of hand. A mid-range line of Macs, labeled Centris,

appeared and disappeared, being integrated into the Quadra product line. Apple introduced a Performa line of Macs, which were identical computers but repackaged for the home market. Mac system taxonomy and nomenclature began to require a scorecard—a very large one at that.

The First-Generation Power Macs

This brings us nearly to the present. Apple was feeling competitive market pressures to lower costs and improve performance. To reduce hardware design costs, Apple standardized most of its computers on the following three basic models.

- The first model uses a low, compact chassis with minimalist expansion capabilities to reduce costs. This design debuted with the Centris 610, followed later by the Quadra 610. It has a single Processor Direct Slot (PDS) that's connected directly to the processor bus. By use of an adapter, the PDS can accept one NuBus board.
- The second model is a desktop configuration that offers three NuBus expansion slots and more capacity for internal peripherals. This chassis was first introduced with the Mac IIvx and was subsequently used in the Centris 650 and Quadra 650 systems.
- The third model is a mini-tower chassis introduced with the Quadra 800 and followed by the Quadra 840AV. Like the second model, this tower system offers three NuBus slots. However, there's ample space for three to four large SCSI hard drives internally, plus a beefy power supply to support them.

All three models have a bay for adding an optional CD-ROM, other removable media drive, or a high-capacity hard drive.

In the area of performance, Apple had been investigating the use of RISC processors in future system designs. This research was evident in products such as Apple's 8×24 GC display board, which uses an AMD 29000 RISC processor to accelerate screen drawing. In addition, the company demonstrated System 7, which was written for the 680x0 processor, running in a software emulator on a Motorola 88000 RISC processor.

Background Info

RISC is the acronym for Reduced Instruction Set Computing. This processor design achieves its high processing speed by implementing many simple instructions. These instructions usually are of a fixed length and execute very rapidly, usually one instruction for every tick of the system clock. This speed is accomplished by limiting what each instruction can do. A handful of instructions, for example, load data from memory to a register, or store data from a register to memory. All other instructions perform fast operations on the contents of the processor's many registers.

These instructions are carefully tailored to minimize overlap between the operations of other instructions. This lets processor designers add execution units—subsections of the processor dedicated to a specific function, such as an integer math unit and a floating-point math unit—that can run in parallel and boost performance by executing two or more instructions simultaneously. As you might expect, simpler instructions require you to use more of them to implement a specific task, so RISC programs typically are larger than Complex Instruction Set Computing (CISC) programs.

We can contrast RISC processors with CISC processors like the Motorola 680x0 and the Intel x86 family. CISC uses variable length instructions to achieve high code density (that is, lots of instructions can be packed into a small amount of memory). These instructions, as their name implies, can perform a sophisticated set of operations and use a wide variety of addressing schemes. One instruction might perform an operation on a location in memory, then step to the next memory location. Another might retrieve a value from memory and then perform a math operation on it.

While some of the simpler CISC instructions can be completed in one clock tick, many cannot. There are several reasons for this. First, because of the variable-sized instructions, the processor is forced to decode the incoming bytes to determine an instruction's length. This takes a clock cycle to perform the initial decode, and then the processor spends additional clock cycles reading in the rest of the instruction. Second, a complex instruction that modifies a memory location requires extra clock cycles to perform the bus operations necessary for the memory access.

continues

continued

Finally, the very complexity of CISC instructions often requires the implementation of a small internal processor—a processor within a processor, so to speak—dedicated to instruction decoding and processor control. This internal processor uses programs called microcode that perform the decode operations. Again, this additional layer of complexity requires extra clock cycles to shuffle instructions through the decoder and operate the microcode that translates the instruction bits into processor actions. Because of RISC's simple instructions, a sophisticated decoder isn't required: You won't find microcode inside a RISC processor. The RISC instruction decoder is implemented completely in hardware and runs at hardware speeds. It takes only several clock cycles at most to translate a RISC instruction into its corresponding actions. A RISC processor's performance is better than a CISC processor's because it can execute more instructions for a given set of clock cycles than the CISC processor.

If RISC technology is so much better than CISC, why is the latter so pervasive on desktop computers? RISC came onto the computing scene much later than CISC. RISC came out of research at IBM, Stanford, and Berkeley in the early 1980s and wasn't commercialized until the middle of that decade. In contrast, Apple Computer sold its first microcomputer, the Apple I, in 1976. By the time RISC processor architecture appeared in the computing industry, CISC processor architecture had been in use for practically a decade.

While CISC has a big advantage in terms of an existing software base, RISC's performance edge should entice users to make the switch. RISC not only allows personal computers to run tasks such as spreadsheets, image editing, engineering simulations, and 3-D image rendering significantly faster, it also provides sufficient horsepower to enable a host of new system services and applications. Some of the possible new system services include a robust, multitasking operating system with memory protection and preemptive scheduling, multimedia services such as MPEG decoding and display, integrated telephony and fax functions, voice and handwriting recognition, and speech synthesis. New applications would be real-time data processing, effortless 3-D image generation and manipulation, and all sorts of multimedia work. For more information on the architectural advantages of RISC, check out Appendix A.

Apple and IBM: Who Could Have Imagined It?

In 1991, Apple teamed up with Motorola and IBM to form an alliance to define the next-generation processor for future desktop computers. Despite the huge legacy of applications composed of CISC code on their respective platforms (Intel x86 code on IBM PCs and Motorola 680x0 code on Macintoshes), they decided that only RISC offered the necessary performance. Cost was an important factor here too. What hindered the acceptance of other RISC systems was the high cost of the RISC processor's fabrication, which in turn resulted in expensive computers.

The alliance is designing and producing a family of RISC processors to be introduced in stages. Each family member is targeted at a specific segment of the computer market. The first family member, the PowerPC 601, was introduced in April 1993. It's targeted at the low-end desktop market, but offers better performance than today's most advanced CISC processor, Intel's Pentium. In October 1993, the alliance introduced the PowerPC 603, a low-power sibling to the PowerPC 601. It is geared toward the notebook market. In April 1994, the PowerPC 604 was announced. Its high-performance design with multiple execution units addresses the mid- to high-range desktop market. The PowerPC 620, introduced in October 1994, is optimized for high-speed transaction servers and high-end workstations.

Over time, faster and enhanced versions of existing PowerPC processors will appear. We've already witnessed some of these enhancements with the 601+, which uses the same 601 circuit design but a new process technology (the complex manufacturing operation by which these chips are made) that both shrinks the size of the circuits and enables the processor to be clocked faster (up to 110 MHz). Another such improvement is the 603e, which was introduced in March 1995 and features larger on-chip caches. It also has some machine architecture improvements. While it uses the same process technology as its predecessor, the 603, the 603e sports a new, faster switching transistor design. Also, certain load/store instructions on the 603e are performance-tuned to operate in fewer clock cycles, and the speed or certain math operations—notably the floating-point divide instruction—were improved. The 603e presently can be clocked at 120 MHz, while the original 603 design topped out at 80 MHz. In August 1995, IBM and Motorola

disclosed a faster version of the 603e, called the 166 MHz 603e. In October 1995, an enhanced 604, the 166 MHz 604e, was disclosed. The same process technology used to fabricate the 601+ was employed to reduce the size and power consumption of these two processors, while boosting their operating speed to 166 MHz. To learn more about the PowerPC family of processors and their features, see Appendix A.

The PowerPC 601 (from now on, I'll just call it the 601) was the heart of Apple's first-generation RISC-based Macintoshes. These systems, mentioned earlier, are called Power Macs to emphasize their performance. There are three systems, and each targets a specific user (see Table 1.1).

Each system is built around one of the three standard model designs discussed earlier. Each Power Mac comes equipped with a base 8 MB of 80 nanosecond SIMM-mounted RAM, a hard drive, built-in Ethernet, and 16-bit stereo sound hardware. The MacOS of these systems contain some important elements. The first is AppleScript, a scripting language that automates repetitive tasks or helps implement custom solutions using several applications. Next, there's the QuickTime Extension for multimedia support. An optional AV Technologies expansion board that provides video I/O and digital video capture can be plugged into the PDS slot on these systems. Bundled with the AV boards is the PlainTalk voice recognition software and the text-to-speech engine.

Table 1.1 An Overview of the First Generation Power Macintoshes

Power Macintosh	*6100*	*7100*	*8100*
Processor	*PowerPC 601*	*PowerPC 601*	*PowerPC 601*
Speed	60/66 MHz	66/80 MHz	80/100/110 MHz
Cache	optional	optional	256 K standard
RAM	8 MB standard	8 MB standard	8 MB standard
DRAM expansion	72 MB	136 MB	264 MB
SIMM slots	2	4	8
Expansion Slots	*One 7" NuBus*	*3 full-size NuBus*	*3 full-size NuBus*

Storage			
Standard HD configs	160MB to 250MB	250M to 500MB	250MB to 1G
Floppy	1.4MB with DMA	1.4MB with DMA	1.4MB with DMA
CD-ROM	Optional	Optional	Optional
Video			
DRAM video	Standard	Standard	Standard
VRAM video		1 MB standard	2 MB standard
VRAM expansion		2 MB	4 MB
Standard support	1 monitor	2 monitors	2 monitors
SCSI	High-speed asynch	High-speed asynch	High-speed asynch Dual SCSI channels
Networking			
	Ethernet on-board with DMA channel, AAUI connector		
Other built-ins			
	16-bit audio stereo in/out with DMA		
	2 Serial ports—LocalTalk with GeoPort		
	compatible with DMA channel		
	Apple Desktop Bus (ADB for input devices)		

The Power Mac 6100/60 takes aim at the low-end user by providing a RISC-based Mac at a low price. It uses the Centris 610/Quadra 610 chassis, and the 601 processor is clocked at 60 MHz. The Power Mac 7100/66 uses the Centris 650/Quadra 650 chassis. With the 601 clocked at 66 MHz and three NuBus expansion slots, this system should meet the mid-range computer user's needs.

The Power Mac 8100/80 stakes out high-end users, with its processor clocked at 80 MHz for best performance. Its Quadra 800/840AV chassis contains ample room for several high-speed SCSI hard drives, and memory can be

expanded up to 264M, which should satisfy the needs of the most demanding power user. Both the Power Mac 7100/66 and the 8100/80 provide a second monitor port, which you can use to expand the screen work area or to run a different operating system on the second monitor. The only second-generation Power Mac that uses a first-generation chassis is the Power Mac 8500/120, which is based on the Power Mac 8100. Power Mac 8100 users can upgrade to this machine with a logic board swap.

The number after the slash in each Power Mac's name denotes the speed of its processor clock. This naming scheme enables faster versions of these Power Macintosh systems to be shipped with the same name because only the trailing digits change. This arrangement eliminates a lot of the confusion created by the previous method in which minor changes to existing Macs begat whole new model names. It also explicitly states the processor speed, which is handy when comparing systems. Such was the case when Apple introduced faster versions of these designs in late 1994 and early 1995. In November 1994, Apple introduced the Power Mac 8100/110, which was basically a Power Mac 8100 system using a 601+ pumped up to 110 MHz. Just two months later, in January 1995, faster versions of the other systems appeared, dubbed the "speed bump" systems because the system clock was increased slightly. There is the 6100/66, the 7100/80, and the 8100/100, with clock speeds of 66, 80, and 100 MHz, respectively. The 8100/110 still exists as a top-notch system for those who need the performance and can afford to pay the premium price it commands.

In May 1995, Apple began selling its first 603-based Mac, the Power Mac 5200. It features a unique chassis that melds a monitor with a stereo sound system, floppy drive, and quad-speed CD-ROM drive into a single unit mounted on a swivel stand (see Figure 1.1).

During the summer of 1995, Apple introduced the Power Macintosh 9500, followed by the Power Mac 8500, 7500, and 7200 series in August. They represent Apple's second-generation Power Macintosh architecture. This architecture features many improvements to boost performance. The memory subsystem uses interleaving on some models to boost memory access rates, and the memory controller is smarter about handling data traffic. The architecture has numerous DMA channels that manage data transfers among the various I/O subsystems such as the network, SCSI, and video capture (on those systems, such as the Power Mac 7500 and 8500, that are equipped with AV technology).

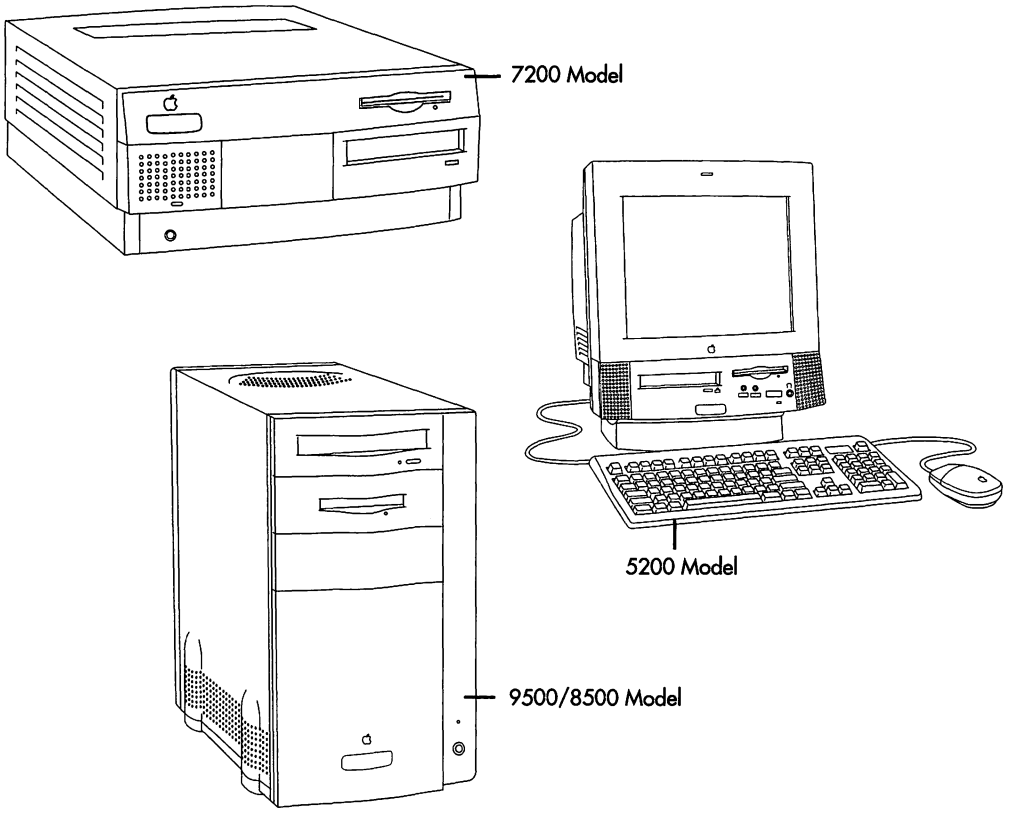


Figure 1.1 The Power Mac 5200 system.

These systems use Open Transport, a Unix standard for network and communications. With Open Transport, Apple uses a standard set of programmer interfaces to implement their network protocol stacks, network interfaces, and serial communications. It also allows future growth paths for supporting PCI network cards such as 100M Ethernet and ATM.

These second-generation systems no longer use NuBus expansion cards. Instead, hardware expansion is handled by another industry standard, the PCI bus. The PCI bus is known for its throughput and plug-and-play capabilities. In fact, a properly designed PCI card is platform agnostic; it can be plugged into either a PC or a Power Mac, and it operates.

Future Directions

The up-and-coming PowerPC CHRP standard allows future IBM and Power Mac systems to run several different operating systems (but not all at once). A crucial capability to support this is that a CHRP system can cold boot into the Mac OS, Windows NT, or OS/2 Warp, and all of the PCI expansion cards in the computer's slots will function with the currently running OS. This is possible because the PCI bus design enables the boot process to detect and load the appropriate drivers from the expansion card's firmware.

The other important capability that allows a CHRP system to host various operating systems is the PowerPC processor is a bi-Endian processor. That is, it supports the two different memory addressing schemes used in the industry. It would be difficult, if not impossible, for a computer to host operating systems that used such disparate addressing modes unless such support was provided in the processor itself. If you're not sure as to what an Endian addressing mode is, relax. You normally won't have to deal with such issues unless you plan to port the code to another platform, such as a PC or Unix system.

The design of the second-generation family Power Macs, like its predecessors, centers around three basic models, as shown in Table 1.2. Each comes equipped with a base 8 MB of 70 nanosecond RAM, a hard drive, two built-in Ethernet ports (one 10BaseT connector, and one AUI connector to handle other types of Ethernet cabling), and 16-bit stereo sound hardware.

Table 1.2 Overview of the Second-Generation Power Macintoshes

Power Macintosh	*7200*	*7500*	*8500*	*9500*
Processor	*PowerPC 601*	*PowerPC 601*	*PowerPC 604*	*PowerPC 604*
Speed	75/90 MHz	100 MHz	120 MHz	120/132 MHz
Cache	optional	optional	256K standard	512K standard
RAM	*8/16 MB standard*	*8/16 MB standard*	*16 MB standard*	*16/32MB standard*

DRAM Expansion	*256 MB*	*512 MB*	*512 MB*	*768 MB*
DIMM slots	4	8	8	12
Expansion Slots	3 PCI	3 PCI	3 PCI	6 PCI
Storage				
Standard HD configs	500 MB to 1GB	500 MB to 1GB	1GB to 2GB	1GB to 2GB
Floppy	1.4M with DMA	1.4M with DMA	1.4M with DMA	1.4M with DMA
CD-ROM	Standard	Standard	Standard	Standard
Video				
VRAM video	1 MB standard	2 MB standard	2 MB	2 MB (via PCI card) standard
VRAM expansion	2/4 MB	4 MB	4 MB	4 MB
Standard support	1 monitor, built-in	1 monitor, built-in	1 monitor, built-in	1 monitor, via PCI card
SCSI	Internal standard	Internal fast	Internal fast	Internal fast
	External standard	External standard	External standard	External standard
Other Hardware				
Accelerated graphics	Composite and S-video		Composite and S-video	
Accelerated graphics	built in	input	input/output	on PCI card
Processor Upgrade				
	Logic board swap	Plug-in	Plug-in	Plug-in
	Ethernet on-board with DMA channel, 10BaseT and AAUI connector			
Other Built-Ins				
	16-bit audio stereo in/out with DMA			
	2 Serial ports—LocalTalk with GeoPort			
	compatible with DMA channel			
	Apple Desktop Bus (ADB for input devices)			

The Power Mac 7200/75, 7500/90, and 7500 use a new low-cost chassis design based on the popular Quadra 630. This chassis resembles a tall, beveled pizza box that provides ample room for several PCI cards and the AV hardware that's present on the Power Mac 7500. Currently, the 7200 series is the low-cost system for the home market. The 7500's built-in AV features and higher performance gear it toward small and mid-sized businesses. The Power Mac 8500 borrows the Power Mac 8100's mini-tower chassis. Its high-speed 604 makes the 8500 useful for the large office, serving the needs of technical and power users. Folks doing multimedia authoring enjoy this system's built-in AV capture and display capabilities. While the 9500's chassis closely resembles the 8100's, it's roughly 2.5 inches taller, to accommodate the six PCI slots and the expansion bays for SCSI peripherals. This generous expansion capacity and fastest 604 processor speed makes the 9500 suitable for high-end publishing and graphics, media authoring, software development, and other jobs where the utmost in performance is required. The 7500, 8500, and 9500 house the PowerPC processor on a plug-in card. This enables you to swap the processor for a faster one when your work load demands it. The hardware in these systems can handle up to a 150 MHz 604 processor in the system.

Important

These second-generation systems use a different type of RAM. First generation Power Macs use 72-pin SIMM-mounted RAM. The second-generation systems use 68-pin DIMM-mounted RAM. If you're planning to upgrade an existing Power Mac to a second-generation system, you'll either have to budget for new memory, or obtain Newer Technology's DIMM Tree, a gadget that lets you protect your memory investment. The DIMM Tree has SIMM sockets that accept your existing RAM, while the device itself plugs into one of the DIMM sockets on the main logic board. Your SIMM RAM must have a 70 nanosecond access time or better to be used this way.

As mentioned earlier, because these computers use a PCI expansion bus, you'll have to replace any NuBus boards that you own.

In August 1995, Apple also introduced a line of PowerPC-based notebook computers. The PowerBook 5300 series is Apple's all-in-one notebook, so named because it can function as a mobile desktop system. It has a large LCD screen (10.4 inches for color screens), a built-in storage bay for a floppy drive or other storage media, a PCMCIA slot that accepts two Type II or one Type III PCMCIA cards (aka PC Cards), a SCSI port, a serial port, and an ADB port—in short, almost a full complement of desktop system capabilities. The Duo 2300 is a light-weight subnotebook computer with minimalist I/O capabilities. It has a RJ-11 socket for an internal modem, and a serial port that can function as an external modem port, printer port, or LocalTalk network connection, depending upon the system settings. For additional I/O capabilities, docking connectors can be attached to the computer, or it can be parked in a docking station. Both PowerBooks use electronic trackpads instead of a mechanical trackball as a reliable pointing device, and a PowerPC 603e processor for high performance and long battery life (see Figure 1.2).

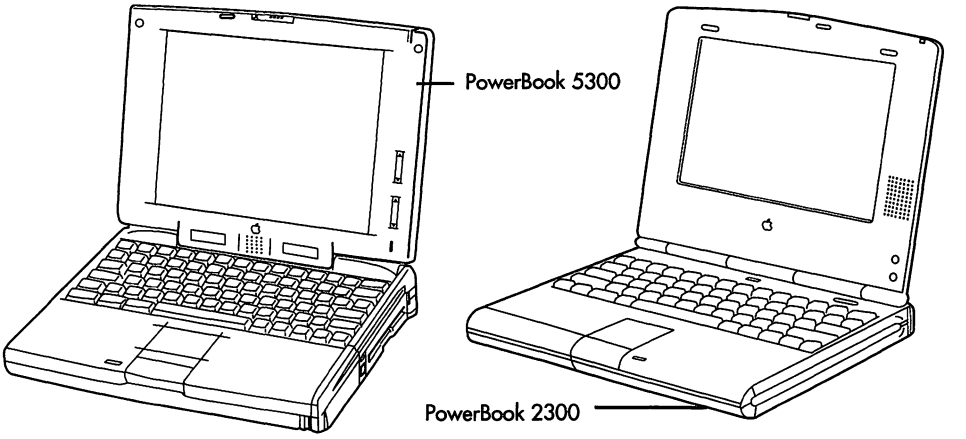


Figure 1.2 The PowerBook 5300 and Duo 2300.

As nice as these systems are, you might suspect that there's a catch, especially regarding software compatibility. After all, didn't Apple and the others sacrifice the existing software base on the altar of performance? Apple tries to let you have your cake and eat it too by placing a 68LC040 emulator in the ROMs of these systems. This emulator is a sort of "virtual" 68040 processor that can execute the 680x0 code in existing Mac applications without modification, but this emulator doesn't support the 68040's floating-point unit (FPU) and memory management unit (MMU) instructions. (Only very

eclectic utility applications would ever try programming a processor's MMU, and such code won't work anyway with the Power Mac's vastly different memory architecture.) The emulator is complete in every other detail, so it can run the bulk of the existing 680x0-based Mac applications and utilities. Lack of an FPU in the emulator might or might not be a problem, depending upon how smart the application software is in dealing with the machine environment. If the application simply expects an FPU, it will crash. Some applications detect the absence of an FPU, and either refuse to run or will do their own computations in software. This slows down the application significantly because such software computations run in the emulator. Those applications that use Apple's math routines will run somewhat faster because portions of these routines were rewritten as PowerPC code.

One reason the emulator works is because of the virtual machine defined by the Mac Toolbox routines. Recall that Mac applications obtain system services (such as reading a file and drawing to the screen) through the Toolbox, and these Toolbox calls act as well-defined entry points into the operating system. What Apple accomplished with the Power Macs was to literally slide a RISC processor into the system and then use "native" (that is, PowerPC) Toolbox code to handle the application's requests. The Mac OS, for example, provides a set of screen drawing primitives known collectively as QuickDraw. An application's drawing functions that use QuickDraw on a 680x0-based Mac continues to work on a Power Mac without recompiling the application. That's because the Power Macintosh ROMs present an identical QuickDraw interface to the application, even though this version of QuickDraw is written in PowerPC code. Whatever application code isn't using the Toolbox gets executed by the emulator.

This is a simplified explanation of the situation, of course. The Power Mac's operating system has to know at any given moment whether it's emulating a 680x0 processor or running native PowerPC code. This is a serious problem because not only is the instruction set different, but the system environment for each processor is different. There are all sorts of system variables, arguments pushed on the stack, and other elements that have to be accounted for when execution switches from the emulated 680x0 processor environment to the PowerPC processor environment and back. A Mixed Mode Manager built into the ROMs along with the emulator manages this context switch. It keeps track of what processor environment the application is currently in, switches the context to the different environment when required, and makes any

necessary adjustments between the two. Such adjustments might pass a drawing request to the native Toolbox code, while another adjustment might communicate the result of the request back to the calling program. For the most part, Mac programmers won't have to concern themselves with how the Mixed Mode Manager works, but there are exceptions. I'll cover them when we get into Power Mac-specific features in Chapters 4 and 5.

For those of you still waiting to hear about a catch in this setup, here it is. The emulator—not surprisingly—musters only the performance of a fast 68030 or slow 68040 processor. Performance varies, depending upon how often the 680x0 application calls the Toolbox routines written in PowerPC code. Because Apple estimates that Mac applications spend 60 to 80 percent of their time in Toolbox code, it's possible that a 680x0 application runs faster than emulated speeds because it spends most of its time actually running native Toolbox code rather than running as emulated 680x0 code. The performance question is complicated by the fact that, for compatibility reasons and time to market issues, Apple hasn't yet ported all several thousand of the Toolbox calls to PowerPC code. 680x0 Toolbox routines that weren't ported get handled by the emulator. In some cases a call to the Toolbox might execute native code, resulting in a brief performance boost, while another Toolbox call might continue through the 680x0 emulator, for a performance hit. It's also important to note that the overhead of the Mixed Mode Manager handling numerous context switches can degrade performance.

So are these Power Macs faster or not? Yes, they're faster. The emulator and Mixed Mode Manager provide compatibility for existing software. They serve as a bridge that allows 680x0 applications to run until the real solution arrives: these same applications written in native code. For such native applications, the overhead of the emulator and Mixed Mode Manager practically disappears, with the exception of those Toolbox routines still implemented as 680x0 code. Over time, applications will run even faster as more of the Mac Toolbox is rewritten as PowerPC code. You can expect future releases of the Mac OS to replace more of the 680x0 portions of the Mac OS with native code, yielding better performance. Early reports, however, indicate that despite the mixture of 680x0 and PowerPC Toolbox code, Mac applications recompiled into native code run very fast on the Power Macs. On the low-end Power Mac 6100/60, such native applications run at

Intel Pentium speeds. These same programs run nearly twice as fast on the Power Mac 8100/80.

The second-generation Power Macs are a clear step further along the path toward rewriting the Mac OS completely in PowerPC code. This version of the Mac OS, System 7.5.2, has some of the Managers, (such as the Resource Manager), some device drivers (such as the SCSI driver), and the network protocol stacks rewritten native code. Those portions of the Mac OS already native (such as QuickDraw, the math libraries, and the Memory Manager) have been tuned for better performance. Finally, the 680x0 emulator has been revised to enhance performance. First, it's been tuned for the 603 and 604 processor architecture. Second, it now uses dynamic recompilation to "compile" frequently executed sections of 680x0 instructions in native code, rather than use the brute-force technique of interpreting each 680x0 instruction over and over again. It does this by monitoring the addresses of 680x0 branch instructions and creates a history table. The history table lets the DR emulator recognize frequently executed sections of 680x0 code (typically code loops). Each 680x0 instruction acts as an index into a look-up table of functions, and each function produces a corresponding set of native instructions. These code blocks get placed in a 512 KB cache and the emulator uses the history table to reroute the course of execution to the cached native code blocks. To minimize overhead, the cache has a simple management algorithm: once it fills, it is purged and the DR emulator begins refilling it. This results in better performance for both 680x0 and native applications because the latter still rely on portions of the Toolbox that are 680x0 code. In terms of performance, the new "DR" emulator boosts application speeds by 15 to 35 percent, on average. Combined with the faster SCSI and smart I/O, these systems are very fast. Even emulated 680x0 applications now readily outperform the fastest 68040 system by a wide margin. However, even better performance is possible by using a native version of the application.

Future Directions

A future version of the Mac OS (code named Copland) will be mostly native code and thus will offer even better performance than is available with the hybrid-code version of the Mac OS today. Its preemptive multitasking kernel will schedule program activity so that the processor is used effectively, realizing even better performance. The kernel's task

scheduler, for example, will transfer execution from a program waiting for I/O (say, a disk read) to one that's processing a complex computation, or to one that needs to write to the network. It also will offer better reliability because Copland will use a limited memory protection mechanism to "wall off" the microkernel, device drivers, and Copland-specific background programs from misbehaving applications.

Time for a Change (to Power Mac)

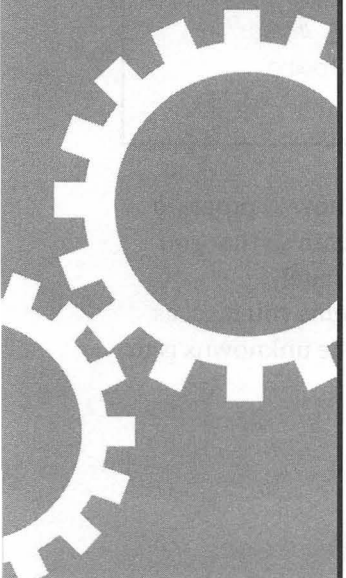
To make the switch to native Power Macintosh applications, programmers need development tools that can compile their existing application code into PowerPC code. Although many different development tools are available, the best possible situation would be tools that run on both 680x0 Macs and Power Macs. Source code that you wrote and tested on a 680x0 Mac could be copied to a Power Mac and easily recompiled, making the initial application port to the PowerPC a snap. (Note: those applications that are fine-tuned to the 680x0 run-time environment will require some adjustments or even a major redesign.) The result is a pair of applications, each of which runs on 680x0 Mac or a Power Mac. With some additional work, you could combine the code in these two applications to make a fat binary application, one that could run on both types of Macs. Or, if the target audience is just Power Mac users, you'd simply write your source code on the Power Mac. Application testing and maintenance would be further simplified if these tools also provided a source code level debugger.

Such development tools exist. It's time for you to meet Metrowerks CodeWarrior, and use it to write some programs.

Beginning Programs

Beginning Programs

This chapter is for the novice programmer. It shows you how to use the ANSI C Standard Libraries supported by the Metrowerks C compiler to do simple tasks on the Mac. The interface for these programs won't be pretty, given that the ANSI Libraries stem from Unix's character-based heritage. The goal here, however, is function, not appearance. These libraries provide a safety net that you can rely on as you explore the Mac's Toolbox and operating system.



As an aspiring Mac programmer, you've no doubt heard this often-quoted maxim about the Mac: "Easy to use, hard to program." Why is this? If you've already leafed through the half-dozen or so volumes of *Inside Macintosh*, you might even know the answer to that question.

Out of this wealth of information, where do you start? Put another way, how do you determine which Toolbox calls to use when starting an application and which ones to call to access services provided by the operating system?

About the Toolbox

The Mac Toolbox and operating system provide more than 4,000 routines, of which about several hundred are commonly used. The Mac is a complex gestalt of these routines and data structures that you must understand fairly well to write a program. How do you know which routines to use? After all, you must understand how to initialize the application's environment so that these routines function, how to plug the application into the operating system so that it coexists and cooperates with fellow applications, and last, but not least, how an event-driven interface works. This seems like a rather dismal attitude to take for a book on Mac programming, but I'd rather you appreciate that there's a lot to learn just to get started in Mac programming than get frustrated and give up entirely.

Background Info

So that you don't get confused later when I start talking about calling c functions, let's make a distinction between those functions our program uses, and those belonging to the Mac Toolbox/OS. Following *Inside Macintosh* conventions, I'll use the term *routine* to indicate Toolbox functions.

Having said that, now I can say it's not impossible to learn how to program the Mac. The trick is to limit the unknowns you're dealing with so that you can break the job into smaller, manageable portions. Fortunately, Metrowerks CodeWarrior provides a way to limit the problems you face, as you'll see in a moment. Another way to deal with some of the unknowns is to

have plenty of source code examples handy. This way you can learn how particular routines operate and when to use them. I'll help you here by supplying some working code examples, which you'll find on the CD.

Munge It

I firmly believe in learning by doing, so let's start by solving a problem. One of my jobs as a technical editor is to take manuscripts and edit them. I clarify certain points in the manuscript, reorganize the flow of thought, request missing material, and perform other editorial tasks. I receive these manuscripts as ASCII text files sent via electronic mail (email) on the Internet or other online services. Ideally, I get a manuscript file and simply start editing it in a word processor. In reality, sometimes there are problems.

Most word processors, both Mac and PC, use a carriage-return (CR) character to end a paragraph of text. This allows the word processor to neatly "wrap" or fit the text on the screen as you add or eliminate words inside the paragraph. Some word processors, however, save the text with CRs at the end of each line. The text looks fine—until you have to change the manuscript using a different word processor. Because of the extra CRs, the word processor can't wrap the words, and you wind up with a mass of jumbled text. The author probably meant well, but the editor now has to laboriously prune those spare CRs from the text, line by line. This type of file is a headache for me to edit.

After hacking away at one long manuscript for over an hour, I decided that this chore was a great job for the Mac to handle. I'd write a Mac program to munge, or hack out, those extra CRs for me. Basically, the program would read an input file, filter out most of the CRs, and write the rest of the data to an output file. Thinking more along the lines of how the computer has to do it, the program reads a byte—or character, actually—from the input file, examines the byte, and if it passes muster (it's not a CR), writes the byte to an output file. If the byte is a CR, it's tossed into the bit bucket instead. If the program detects the end of a paragraph (a double CR, or a blank line), then the end of paragraph (the double CR) is written to the output file. This makes the resulting output ASCII text organized the way a word processor expects it. Stated this way, the problem seems easy enough.

Now here's where CodeWarrior helps. Metrowerks CodeWarrior supports the ANSI C Standard Library, which is based on the Unix C function libraries. These libraries supply functions that handle file I/O and provide an interactive console where you enter commands and get screen output. Because these functions originally were implemented on old Unix systems, they typically deal with character-based I/O. This doesn't make for a nice Mac interface, but it lets you concentrate on the problem without having to learn lots of Toolbox routines all at once.

Important

CodeWarrior's console I/O provides support for the C Standard Library's `stdin`, `stdout`, and `stderr` streams. It opens a virtual console window where all these streams are directed. The console window is set up and managed by CodeWarrior's SIOUX (Simple Input/Output User eXchange) library, which must be linked to an application. The CodeWarrior IDE can automatically include the SIOUX and several other C libraries in a new project file, sparing you the trouble of adding them yourself. It does this by drawing information from a Stationary file that you pick when creating a new project. Stationary files act as templates. These templates have built-in references to the most often-used libraries for certain types of projects. Most of the Stationary files include the ANSI C libraries by default. If you don't want to use the ANSI C libraries, pick one of the "minimal" Stationary files when you create a new project.

Getting Started

Let's get started by launching the CodeWarrior C compiler, or more accurately, the application that manages it. The easiest way to do this is to go inside CodeWarrior folder, open the Code Examples PPC folder, followed by the Munger folder, and double-click on the file "munger.c." After a short delay, a window opens, displaying C code. This window belongs to the built-in editor that's part of CodeWarrior's Integrated Development Environment (IDE). The CodeWarrior IDE application hosts and seamlessly combines all the development tools that you'll need—editor, C compiler, resource compiler, and linker—to write Mac programs. Within the editor's window, you should see the following code:

```
#include <stdio.h>

#define CR 0x0D
#define LF 0x0A

FILE *istream, *ostream;

void main (void)
{
short  crflag;
long   icount, ocount;
char   ifile[64], ofile[64];                /* Path names must be 64 chars or less */
int    nextbyte;

printf ("Enter input file: ");
gets (ifile);
if ((istream = fopen(ifile, "rb")) == NULL)          /* Open the file OK? */
    {
    printf ("\nError opening input\n");              /* NO, say so */
    return;                                          /* Bail out */
    } /* end if */

printf ("Enter output file: ");
gets (ofile);
if ((ostream = fopen (ofile, "wb")) == NULL)         /* Can we write an output file?*/
    {
    fclose (istream);                                /* NO. First close input file */
    printf ("\nError opening output\n");             /*   then warn, and bail out */
    return;
    } /* end if */

icount = 0L;        /* Set counters */
ocount = 0L;
crflag = 0;

/* Read char.s until end of file */
while ((nextbyte = fgetc (istream)) != EOF)
    {
    icount++;                                        /* Bump input char counter */
    switch (nextbyte)                                /* What char was read? */
        {
        case CR:
```

```
        if (crflag >= 1)                        /* Two in a row, end of paragraph */
            {
            fputc(nextbyte, ostream);           /* Write two CRs to the output */
            fputc(nextbyte, ostream);
            crflag = 0;                         /* Reset the flag */
            ocount++;
            } /* end if */
        else
            crflag++;                           /* Bump the flag, and toss the CR */
        break; /* end case CR */
        case LF:                                /* Toss LF, but don't touch crflag */
        break; /* end case LF */
        default:
            fputc (nextbyte, ostream);          /* All other chars get written */
            ocount++;
            crflag = 0;                         /* Clear the flag */
        } /* end switch */
    } /* end while */
fclose (istream);                               /* Clean up */
fclose (ostream);
printf ("Bytes read:    %ld\n", icount);
printf ("Bytes written: %ld\n", ocount);
} /* end main () */
```

Let's take a closer look at this code.

The Code Tour

The munger program first prompts for an input filename, using the `printf()` function to put a message in a console window made by the C Standard Library. It uses `gets()` to read the keyboard when you type in a filename and press Return. Your input is placed in the array `ifile`. Note that `ifile` and `ofile` are 64 characters long. If you're opening files with long names, or the file is in a folder with a long name, you need to increase the sizes of the `ifile` and `ofile` arrays so that the pathname fits.

Background Info

A pathname is the complete description of where a file is located on a particular hard drive or volume. It's a string of characters that incorporates the volume's name, the filename, and the names of those folders within

which the file is nested. This might sound overly complicated until you realize that you might have two files named "Résumé" on the same hard drive, but in different folders. As an example of a directory pathname description, consider the following. If a Mac's hard drive is named Tachyon, and a file "Read Me" is in the folder New Info, the pathname for the file is `Tachyon:New Info:Read Me`. Another "Read Me" file, located in the folder named StuffIt, that in turn is inside another folder called Aladdin Docs, would have the pathname `Tachyon:StuffIt:Aladdin Docs:Read Me`. This convention is similar to DOS/Windows pathnames, but instead of a backslash (\), the Mac OS uses colons as separators between the drive, folder, and filenames. This convention also explains why you can't use a colon in a filename.

Next, the program uses `fopen()` to open the file:

```
if ((istream = fopen(ifile, "rb")) == NULL)          /* Open the file OK? */
    {
    printf ("\nError opening input\n");              /* NO, say so */
    return;                                          /* Bail out */
    } /* end if */
```

Note that you check to see if this open operation fails. If it does fail, the program halts. With the minimalist input provided by the C Standard Library, it's quite possible for you to mistype the filename, which creates an error condition when `fopen()` fails to open the file. The program then uses similar code to set up the output file and checks for trouble as it does so. This is a good time to emphasize that no matter how simple or complex your program is, always, ALWAYS, *ALWAYS*, check for errors. You can eliminate a lot of crashes, trashed hard disks, and needless debugging by having your program determine if the routines it calls complete successfully.

The heart of the program is the `while` loop, which reads a stream of bytes from the input file and processes them. The `switch` statement inside the loop determines the fate of the byte under scrutiny. Any character other than a CR or linefeed (LF) falls through to the default `case`, which writes the character to the output file. Because I get lots of files from PCs, and DOS ASCII text files use a LF-CR combination to end each line, the program also filters out any

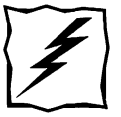
LF characters it happens to find in the character stream. The program handles this filtering operation with the LF `case` statement, which simply does nothing, and as a consequence the LF never gets written to the output file.

Now to those CRs, which are handled by the case statement:

```
case CR:
if (crflag >= 1)                    /* Two in a row, end of paragraph */
    {
    fputc(nextbyte, ostream);       /* Write two CRs to the output */
    fputc(nextbyte, ostream);
    crflag = 0;                     /* Reset the flag */
    ocount++;
    } /* end if */
else
    crflag++;                       /* Bump the flag, and toss the CR */
break; /* end case CR */
```

The program logic works on the assumption that most folks separate paragraphs with a blank line. This means that the last line of the paragraph ends with a CR, which is followed immediately by a blank line composed of a second CR. So when the program encounters the first CR character, it gets tossed into the bit bucket and the flag `crflag` is incremented. If a character other than CR is read next, the program clears `crflag`. This handles situations where the CR just terminates a line of text. Notice the exception here: A LF character doesn't reset `crflag` because it occurs jointly with the CR in DOS files. When a second CR in a row occurs because of a blank line, the `if` statement detects that `crflag` is set. The code now writes two CRs to the output file to ensure the line break between paragraphs. Of course, we clear `crflag` to begin the search for the next paragraph ending.

Finally, the program closes both files and writes a summary to the console window of the bytes read and written, as tallied by the counters `icount` and `ocount`. Because the program's function is to throw away bytes, fewer bytes should have been written than read. It's not necessary to do this, but the summary serves as a sanity check on the program's operation, which is reassuring to me. It's possible to defeat the paragraph detection logic by submitting an ASCII text file with no blank lines between each paragraph, but I can add 30 to 70 blank lines to a manuscript within minutes, while manually stripping CRs from over several hundred lines takes up to an hour.



Important

The text of this book was written using the full version of Metrowerks CodeWarrior. You'll have to use slightly different steps when using CodeWarrior Lite from the CD. The commands New, New Projects, and Add File... are not available. Because of these limitations, it can only work with the sample files provided on the CD.

So, if you are following along using CodeWarrior Lite, when the text tells you to use the New, New Project..., or the Add File... command, you should instead open the related project file and keep it open throughout the exercise. All the associated files will already be in the project, so you won't need the Add File... command. Then, you can follow the same procedures as if you were using the full version of CodeWarrior.

Making Munger

Let's make this file munging program. You've opened the file "munger.c," so the next step is to make a project file for it. From the File menu, select New Project..., type Munge.μ (you get the μ character by typing Option-M) for the project name into the Standard File dialog box that appears.

There's an informal convention where you denote a project file by attaching either a .μ or .π, or .prj extension to the filename. Use of the .π extension (the character is made by typing Option-P) is common among THINK C programmers, while CodeWarriors favor the .μ extension (as mentioned, made by typing Option-M). This naming convention isn't required, however. You can use any extension (or none) if you want, but if you're working with other programmers or plan to share code with other users, these conventions help identify the project file for them.

Take note of the Project Stationary item that appears in Standard File dialog window. This popup menu lets you pick from various Stationary files. These files preconfigure the settings of the project file that you're making. Hold down the mouse button on this popup menu and pick the item Mac OS PPC C/C++.μ. Now hit Return or click the Save button.

Now a project window appears, and its window title matches the name of the project file you just entered. This project window displays and manages the

several types of files that make up the project, and ultimately, a Mac program. There's a Source entry for source code files, a Resources entry keeps track of the resources associated with the project (if any), and there are entries for the program's libraries, which are divided into Mac OS libraries and ANSI libraries. In some of these categories, default filenames are provided. It supplies InterfaceLib, MathLib, and MWCRuntime.Lib, for example, as the default Mac OS libraries. The project also has set up some default ANSI C libraries, whose entries you'll change in a moment. First, click on the Sources entry to highlight it. Now choose Add File... from the Project menu. In the Standard File dialog box that appears, locate the file "munger.c" and click on the Add button (see Figure 2.1).

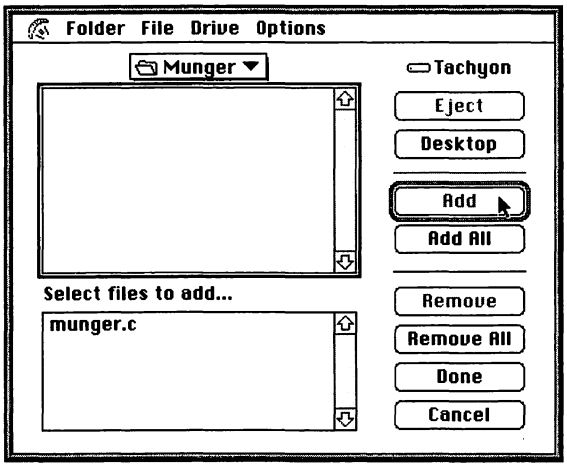


Figure 2.1 Adding "munger.c" to a project.

The Add button dims, and next you click on the Done button. The source file "munger.c" is added to project Munger.μ. You added this one file, and—surprise!—it's time to delete some surplus entries from the project. First, select the <replace me Mac>.c entry in the window's Sources category by clicking on it. Next, go to the Project menu and pick Remove Files. Repeat these steps for the <replace me>.rsrc from the Resources category. You removed these items because they simply acted as placeholders for your own files. Also, your first programs won't be using resources anyway. (If you're wondering what a resource is, hold tight until Chapter 4.) Now go to the ANSI Libraries category and click on the arrow to expose the files kept in this

section. Highlight the entry ANSI C++PPC.Lib and again chose Remove Files. After you've made these changes, the Project window appears as shown in Figure 2.2.

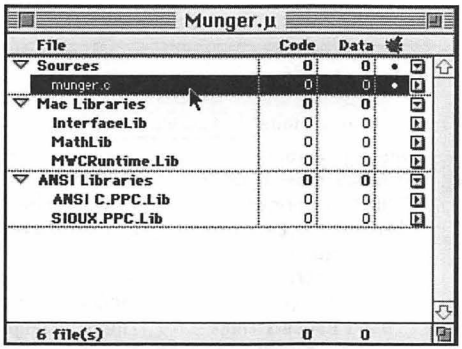


Figure 2.2 Changing the library files in project Munger.µ.

You're not done yet. Select Preferences... from the Edit menu. Scroll to and click on the C/C++ group icon. In the panel that appears, ensure that the checkbox for Require Function Prototypes is set (see Figure 2.3). This setting demands that you declare each function, specifying the function's number and type of input arguments and the type of the result (if any). This can catch potential problems that can occur when you call the function with a set of arguments different from what it expects. This might happen because you're modifying the function, or inadvertently passed the function an argument of the wrong type, as when you call a Toolbox routine. In either case, checking Require Function Prototypes nails this error at compile time. Otherwise, when the program runs, such improper function calls might cause a crash. I also delete the MacHeaders.h precompiled header filename from the Prefix File Item because my work often involves parts of the Mac OS that aren't normally in the precompiled header file.

Next, go to the C/C++ Warnings group and click on Unused Variables, Unused Arguments, and Extended Error Checking. Like Require Function Prototypes, you actually don't need these settings for this project, but because they enforce good programming practices, you ought to get into the habit of setting them now. The unused variables/arguments settings typically catch "dead code," such as a local variable whose code you eliminated from a function, but forgot to remove its storage declaration. Extended Error

Checking uses stricter type-checking rules when compiling the C code, flagging subtle code goofs. The C compiler, for example, issues a warning if a non-void function doesn't have a return statement, or a value isn't passed to the `return` statement.

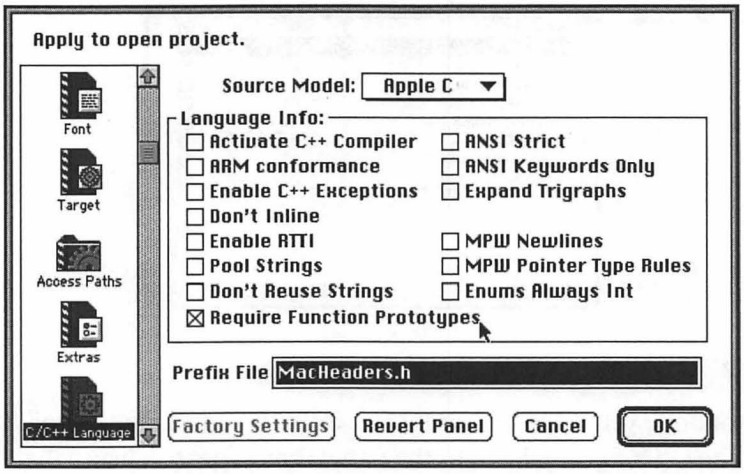


Figure 2.3 Setting the C/C++ preferences for project Munger.μ.

Now scroll to the PPC Linker group icon and click on it. In this group's panel, go to the Entry Points section. We're just going to check the default functions that get called when our program initializes, starts, and exits. These functions, which are part of the Power Mac run-time architecture, get called when a native program launches and quits. Both the Initialization and Termination items in this panel should be blank (see Figure 2.4). The Main item has a function name of `__start`. This function is responsible for calling our program's `main()` function. This is the default situation for an application; it usually doesn't require any special initialization or termination processing. The Mac OS normally handles any set up or house-cleaning when your application launches and quits. For shared libraries, the Initialization and Termination items have the function names `__initialize` and `__terminate`, respectively. When the Mac OS loads the shared library, it calls these functions to handle any special processing the library requires, such as allocating and subsequently releasing a large block of memory.

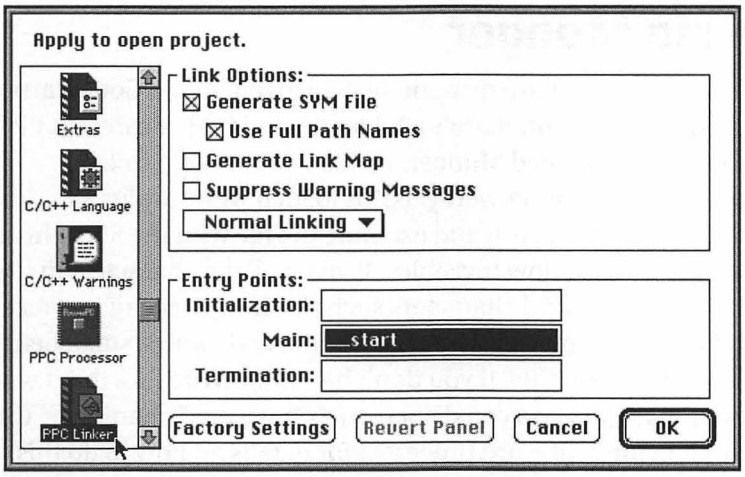


Figure 2.4 Checking the entry points for the project.

Finally, go to the PPC Project panel and type **Munger** for the application name into the File Name text box (see Figure 2.5) and click on the OK button. Click on the Toolbar's Make button or select Make from the Project menu, and let CodeWarrior go to work on the project. If there are no problems, processing statements from the compiler and linker briefly appear in the Toolbar's status area. An application named Munger is created.

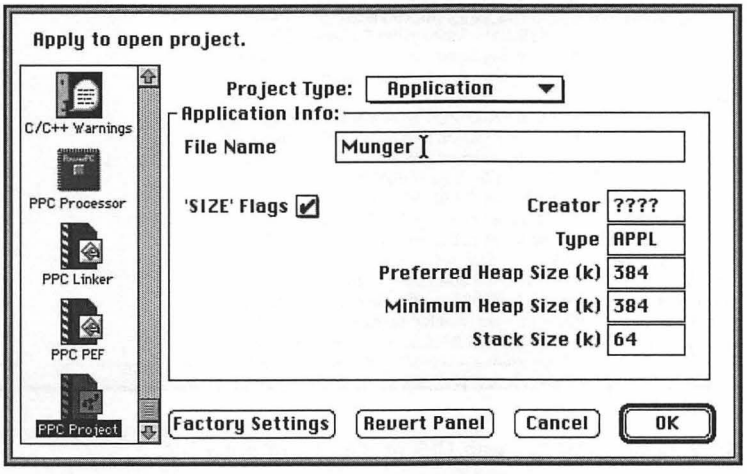


Figure 2.5 Setting the name of the application file that the project makes.

Running Munger

Suppose that on a Mac hard drive named Tachyon, in the CodeWarrior tools folder called CodeWarrior, there's a folder named Code Examples PPC, followed by a folder named Munger. Inside it is a text file called "PowerPC.txt." Suppose "PowerPC.txt" is loaded with surplus CRs. First, open the file in MacWrite Pro and examine the file with the Show Invisibles set in the View menu. Show Invisibles displays all the characters in the file—including invisible control characters such as CR—instead of just text characters. In Figure 2.6, you can see that each line ends with a small bent arrow symbol; they represent CRs. If you don't have MacWrite Pro, don't worry: other word processors also can display such "invisible" characters. Check the documentation for your word processor for details on how to do this.

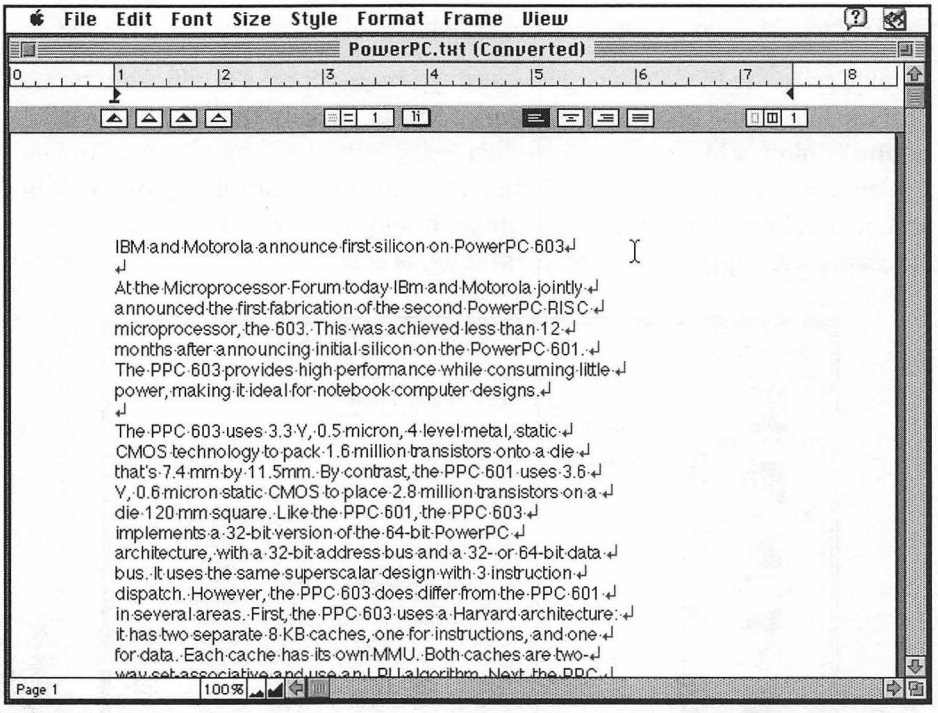


Figure 2.6 A sample text file, with CRs at the end of every line.

It's time to set "munger" to work on this file and see what happens. Launch "munger" from the CodeWarrior IDE by clicking on the Run button in the Toolbar. A console window called munger.out appears. Type in the

pathname to the sample text file we examined earlier as follows: `Tachyon:CodeWarrior:Code Examples PPC:Munger:PowerPC.txt`. Of course, if your hard drive name and CodeWarrior tools folder are named differently, you'll type the appropriate names into the pathname. If you goof on the filename, "munger" complains and the program stops. If the filename is OK, "munger" asks for an output filename. Type a filename that uses the same file path, such as `Tachyon:CodeWarrior:Code Examples PPC:Munger:PowerPC.out`. Press Return and "munger" processes the file. You'll get a summary of the operation, as shown in Figure 2.7. The munger.out console window remains present, and you have to pick Quit from the File menu to leave "munger." When you do so, a dialog box appears that asks if you want to save munger.out's contents. Click on the Save button if the console window's output is important to you. Otherwise, click on Don't Save to discard the console window's output. This feature enables you to capture the output of a job as required. For lengthy pathnames, as in the example, the SIOUX console window lets you copy and paste characters. You only have to type in the pathname once for the input file prompt, select this text with the mouse, copy, and then paste the bulk of the pathname into the prompt for the output file pathname. Now all that's left is for you to type the name of the output file.

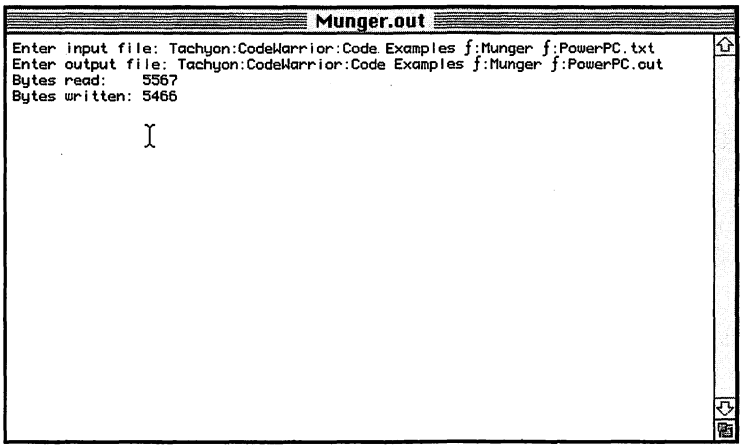


Figure 2.7 The console window of program "munger" after it processes a file.

Now if you open the resulting file "PowerPC.out" with your favorite word processor, you'll see that "munger" did handle the surplus CRs (see Figure 2.8).

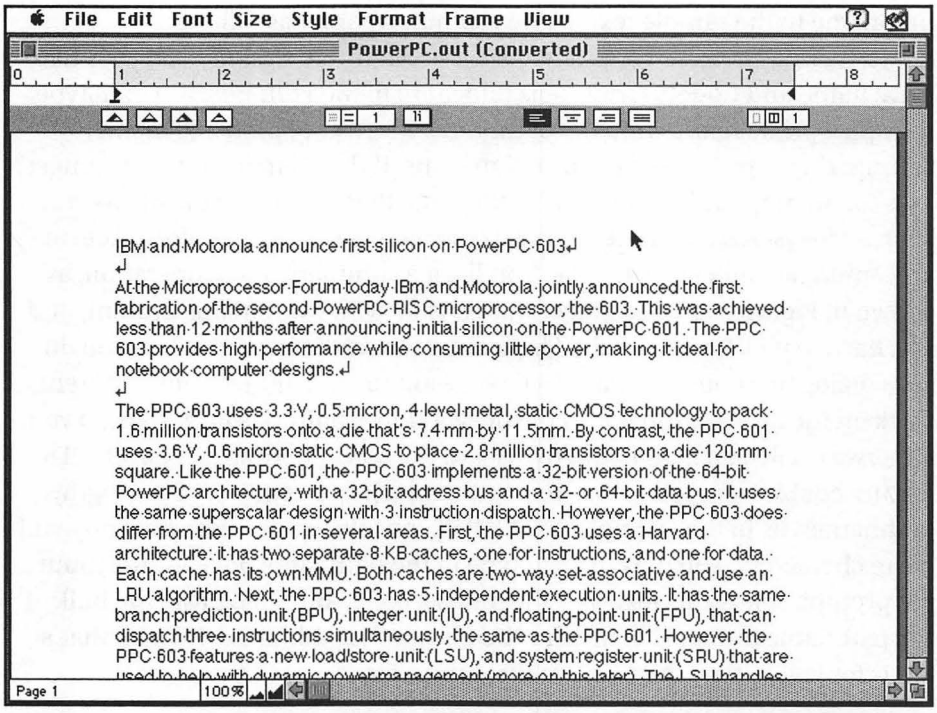


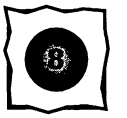
Figure 2.8 The munged output file.

Where's the Mac?

OK, so you got some C code to run on the Mac, but where is that easy-to-use Mac interface? The point is that we got code running quickly without getting mired in too many details. We let the C Standard Library handle the job of initialization. It also provided I/O through a Mac window masquerading as a console window. The important thing to carry away from this exercise is that you can use the C Standard Library to act as a scaffolding while you test various algorithms and Toolbox calls. The programs you make this way aren't meant to be friendly, just useful enough to test code. We will start adding our own Mac interface to our "munge" program in the next chapter.

Here's another example where the C Standard Library pitches in while we investigate some Toolbox routines. Under System 7, active applications are called processes. Certain system services such as File Sharing, PlainTalk voice recognition, the Express Modem, and the LaserWriter 8.3 background

Desktop Printer Spooler actually are processes themselves. These system services don't show up on the application menu, but they do operate quietly in the background. As the Mac migrates to a preemptive multitasking OS, processes will become even more important to the overall operating system design. With that in mind, let's take a closer look at processes.



Future Directions

The future Mac OS that I'm talking about here is Copland. Copland's microkernel, drivers, and specialized background applications will run in a separate memory space, protected from a malfunctioning application. Programs that make use of the GUI must run together in another memory space. This is because the QuickDraw code currently is non-reentrant (for more information on reentrant code, see Chapter 4). Most of the processes just described, however, don't use QuickDraw, and thus are ready to take full advantage of the capabilities that Copland offers. But don't some of these background services, such as PlainTalk and File Sharing, use Control Panels, which in turn use QuickDraw? Yes, they do. However, each Control Panel is simply a front end that, based upon the controls you set, sends the appropriate messages to the background processes that do the actual work.

Processes Revealed

The Mac OS allocates each process a partition in memory where it runs and assigns it a unique ID number. This ID number is called the process serial number (PSN) and it is used by the operating system to reference the process and control it. *Inside Macintosh: Processes* documents a group of Toolbox routines, known collectively as the Process Manager, that manage these processes and supply information on them. To find out more about processes, let's examine another quick program. Go to the Code Examples PPC folder, and open the Process folder. Double-click on the "process.c" file.

```
#include <processes.h>
#include <stdio.h>

void main (void)
{
```

```
register int            i;
ProcessInfoRec          thisProcess;
ProcessSerialNumber     process;
FSSpec                  thisFileSpec;
unsigned char           typeBuffer[5] = {0};
unsigned char           signatureBuffer[5] = {0};

thisProcess.processAppSpec = &thisFileSpec;             /* Aim pointer at our storage */
thisProcess.processInfoLength = sizeof(ProcessInfoRec); /* Store record size */
thisProcess.processName = (unsigned char *) NewPtr(32); /* Allocate room for the name */
process.highLongOfPSN = kNoProcess;                     /* Clear out process serial number */
process.lowLongOfPSN = kNoProcess;

while (GetNextProcess(&process) == noErr)               /* Loop until all processes found */
    {
    if (GetProcessInformation(&process, &thisProcess) == noErr)  /* Obtain detailed info */
        {
        for (i = 0; i <= 3; i++)        /* Copy type & sig info into string buffers */
            {
            typeBuffer[i] = ((char *) &thisProcess.processType)[i];
            signatureBuffer[i] = ((char *) &thisProcess.processSignature)[i];
            } /*end for */
        printf ("Process SN: %ld, %ld, Type: %s, Signature: %s, Name: ",
            thisProcess.processNumber.highLongOfPSN,
            thisProcess.processNumber.lowLongOfPSN,
            typeBuffer,
            signatureBuffer);
        printf (" %s \n", P2CStr(thisProcess.processName));  /* Now print the name */
        } /* end if */
    } /* end while */
} /* end main() */
```

This program uses the Process Manager to obtain information about all of the processes running on the system. Notice that we include one more header file, `<processes.h>`, to the source code. This header file defines the Process Manager routines and a data structure called `ProcessInfoRec` that acts as a container for all of the process' relevant information. The lines

```
thisProcess.processAppSpec = &thisFileSpec;             /* Aim pointer at our
➥storage */
thisProcess.processInfoLength = sizeof(ProcessInfoRec);  /* Store record size */
thisProcess.processName = (unsigned char *) NewPtr(32);  /* Allocate room for the
➥name */
```

```
process.highLongOfPSN = kNoProcess;                    /* Clear out process
➥serial number */
process.lowLongOfPSN = kNoProcess;
```

are used to set up our local copy of `ProcessInfoRec`, called `thisProcess`. Then we direct pointers in `thisProcess` to the appropriate storage locations. `processAppSpec`, for example, which contains the location of the file that created the process, is aimed at `thisFileSpec`. And `processName`, which holds the process' name, is directed to a chunk of memory allocated by `NewPtr()`, a Toolbox memory allocation routine. Last, we clean out the PSN variables by assigning `kNoProcess`, which equals zero, to it.

Now we use a `while` loop that calls the Process Manager routine `GetNextProcess()` repeatedly. `GetNextProcess()`, when called with a PSN of 0, starts at the beginning of an internal list of PSNs maintained by the Process Manager and returns the first PSN on the list. By passing each returned PSN back to `GetNextProcess()` on subsequent tours of the loop, we walk this list and use another routine, `GetProcessInformation()`, to grab information on every process in the system. When `GetNextProcess()` finally reaches the end of the PSN list, it returns an error value and the loop completes.

While the loop cycles, `GetProcessInformation()` extracts in-depth information on the current process and stuffs it into `thisProcess`. As usual, notice that we check for errors. If `GetProcessInformation()` reports no errors after it completes, we dump some of the information it gathered to the console window.

Gathering Processes

It's time to compile the "process.c" program and see what it gathers. There are seven steps, and they are nearly identical to the first program, "munger."

1. Save the code (if you typed it in) into a file called "process.c."
2. Create a new project called process.μ. In the Standard File dialog window, be sure to pick the Stationary file Mac OS PPC C/C++.μ from the popup menu. Add "process.c" to the project. Check that the usual suspects, "InterfaceLib," "MathLib," "MWCRuntime.Lib," "ANSI C.PPC.Lib," and "SIOUX.PPC.Lib," are in the project. The Project window should resemble Figure 2.9.
3. Set the PPC Language and PPC Warning preferences the same way you did for the `Munger.μ` project.

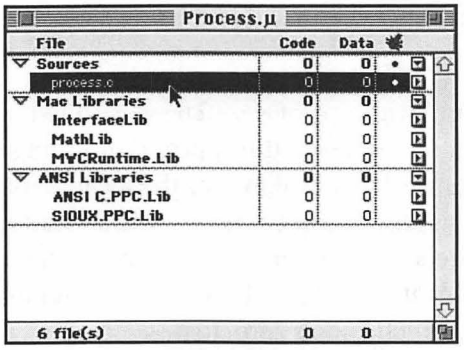


Figure 2.9 The Project window for the process program.

4. In the PPC Linker preferences panel, check the entry point settings. As mentioned previously, the defaults for this program are fine, but you should get into the habit of visiting this panel when we start writing more capable Mac programs.
5. Name the output file `Process` in the Project preferences panel.
6. With all the preferences set, make the program.
7. Finally, pick Run from the Project window. The console window appears and displays information on each process' PSN, type, signature, and name (see Figure 2.10). Note the presence of our own program, "Process," as well as the CodeWarrior compiler, the Finder, the File Sharing Extension, and other applications.

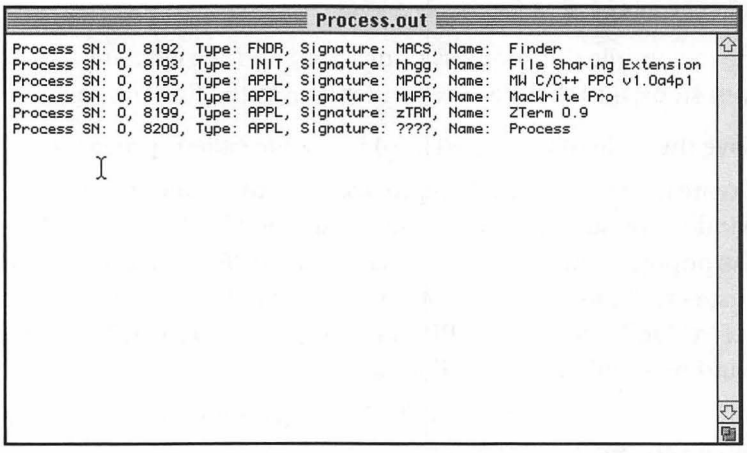


Figure 2.10 The process program displaying all processes on the system.

A Word of Caution

As you can see, with the assistance of the I/O functions provided by the C Standard Library and SIOUX, you easily can delve into the Mac's inner workings. Even with all the Mac code I've written over the years, I still frequently use the C Standard Library I/O functions to quickly test code that uses unfamiliar Toolbox routines. If you know that you're going to be using the ANSI libraries exclusively in your program, pick the Stationary file ~ANSI PPC C/C++.μ from the popup menu when creating the project file.

Hazard

Because the C Library, through SIOUX, does its own application initialization, you need to exercise caution when mixing this library with certain Toolbox routines. For example, the `printf()` function has SIOUX create a Mac window that mimics a console window. If your program happens to initialize the Window Manager so that it can use a Toolbox routine, this creates a situation where your initialization code butts heads with the window data structures created by SIOUX, and causes a crash.

To avoid this pitfall, never match the I/O functions you use with the Mac Toolbox with those of the C Library in the program. If you use QuickDraw or Window Toolbox routines in your code, don't use the C Library functions that require a console window. Or, if your program uses the C Library's file I/O functions, don't use Mac Toolbox's file I/O routines.

In CodeWarrior 1.3 (aka CW7), you can modify the behavior of SIOUX such that you can actually initialize parts of the Mac Toolbox, add custom menus, and even install your own event loop handler. To do this, you need the include header file "SIOUX.h" in your source code, and change some fields in a `SIOUXSettings` data structure. Here's an example where you set up SIOUX to allow custom menus and your own event loop in a program:

```
#include <stdio.h>
#include <sioux.h>

void main(void)
{
SIOUXSettings.standalone = FALSE;
SIOUXSettings.setupmenus = FALSE;
```

continues

continued

```
.
.
<program code>
.
.
} // end main()
```

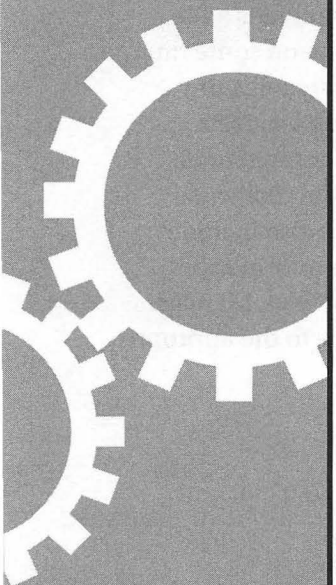
Tread carefully, and consult the CodeWarrior documentation for more information.

Just the Beginning...

In this chapter, you've seen how to build a practical application, leveraging off the I/O functions in the C Standard I/O Library. We've outlined seven steps required to build and run the application in Metrowerks CodeWarrior. You also examined how to use the C Library to help us experiment with various Toolbox routines in isolation. Now you can apply this knowledge to learn how the Mac works, which ultimately assists you in writing Mac applications. Try some experiments of your own, and then proceed to the next chapter where you'll write a full-blown Mac application.

Using the Toolbox

At this point you should be comfortable with the Metrowerks CodeWarrior integrated development environment and how to create and manage a project. In a jam, you can rely on CodeWarrior's C Standard Library to help you learn how to use new and unfamiliar Mac Toolbox and OS routines. Does this mean you're ready to write a full-fledged Macintosh application? Not quite. For novice Macintosh programmers, there are a number of basic concepts to learn. These include program initialization, resources, event handling, and the structure of files. These concepts cover a lot of ground, but I'll keep the information doses manageable by introducing them in stages, along with programs that demonstrate these aspects of the Mac OS. Readers with intermediate Mac expertise might want to jump to the back of the chapter and study the code on Apple Events. The rest of us will catch up with you later.



In Chapter 2, I mentioned a Process Manager. As you learned, it is a collection of routines that deals with processes, which are running applications. It should come as no surprise that many of the Toolbox routines are organized into groups of related functions, or Managers. The Event Manager deals with low-level events such as mouse clicks and keystrokes. A Memory Manager has routines that allocate memory, release memory, and adjust the size of the stack. A Window Manager provides routines necessary for the care and feeding of windows, while a Font Manager deals with the various fonts you see on the screen or use to print. The list goes on and on. One of the few exceptions to this naming scheme is QuickDraw—the routines that handle drawing on the screen or onto a page image bound for the printer. These various Managers serve as libraries of routines available for your use.

Important

The Toolbox routines actually exist either as shared libraries or as functions in the Power Mac's ROM. For 680x0-based Macs, entry into a particular Toolbox routine is handled by a 680x0 processor exception, which then jumps to a dispatch table. For the Power Macs, a Toolbox routine is entered via a set of pointers called a transition vector. These transition vectors are set up by the operating system when an application loads. If you're wondering what an exception or transition vector is, never fear: the details are explained in the next chapter.

What's nice about this scheme is that it helps organize all of those thousands of Toolbox routines. If, for example, you need a function that reads a file, look at the File Manager routines. As a novice, you should spend some time just browsing through *Inside Macintosh.* The new editions organize the technical content by category, such as files, memory, text, and so forth, rather than by volume number as they did in the past. This arrangement helps you locate the various Managers by function. Along with the usual reference information, the new editions of *Inside Macintosh* also include some tutorial material. You might not understand all of the information presented there (for now), but it will give you a good idea of what Managers exist, and what they do. When necessary, I'll make reference to the appropriate *Inside Macintosh* edition.

Meet Some Managers

To get you used to the idea of Managers, start by writing the classic "Hello world" program. This also will demonstrate how to initialize a Mac application. Start by opening the Code Examples PPC folder. Now open the MacHello folder and double-click "hello1.c." Now let's take a close look at the code:

```
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Windows.h>
#include <Memory.h>
#include <Events.h>
#include <OSUtils.h>


#define NIL             0L
#define IN_FRONT        (-1)
#define IS_VISIBLE      TRUE
#define NO_CLOSE_BOX    FALSE
```

Already you'll notice that there are a lot more header files involved than just using the Standard C Library's `<stdio.h>`. That's because the Standard C Library includes every I/O function possible plus the kitchen sink. In contrast, each Toolbox Manager has a separate header file. This keeps both your workload and the compiler's at a manageable level. It means that you have to be more aware of what routines you plan to use (yet another reason to browse through *Inside Macintosh*).

Background Info

Like Symantec's THINK C, the Metrowerks CodeWarrior compiler uses a special header file called "MacHeaders.h" whether you're generating 680x0 or PowerPC code. This file incorporates the most frequently used header files, such as "QuickDraw.h," "Fonts.h," "Windows.h," "Files.h," and others. "MacHeaders.h" references a precompiled processor-specific header file, either "MacHeaders68K" or "MacHeadersPPC", depending upon which processor you've selected in CodeWarrior's Target preferences panel. This precompiled header file helps boost the compiler's processing speed when it searches for routine definitions. It also means that if you stick with the

continues

continued

most frequently used Manager routines, you needn't worry about typing `#include` statements. However, not all of the header files are incorporated into MacHeaders. If you're using some of the more sophisticated Toolbox routines to, say, play sounds or do special printing, you'll need to include those files at the start of your source code. Or, you can edit the supplied "MacHeaders.c" file to add the missing header files, and recompile it with the CodeWarrior compiler. This makes the header files you need part of the precompiled header file. The "MacHeaders.c", "MacHeaders.h", and assorted support files are located in the MacHeaders folder.

I prefer to enter all the header files anyway. This way you keep better track of what Managers you're using, which helps with your program design. It doesn't hurt having the header files declared in your program, because even if you use the "MacHeaders" file, the Metrowerks CodeWarrior compiler is smart enough to sort things out and prevent redundant declaration errors from cropping up.

The definitions `NIL`, `IN_FRONT`, `IS_VISIBLE`, and `NO_CLOSE_BOX` are for use later in the program. As you'll see, they'll make a Window Manager routine that we use a lot easier to understand. Now enter:

```
void main(void)
{
WindowPtr   thisWindow;
Rect        windowRect;

/* Lunge after all the memory we can get */
   MaxApplZone();
   MoreMasters();
   MoreMasters();

/* Initialize the various Managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitCursor();
```

Now we're getting somewhere. The variable `WindowPtr` holds a pointer to a data structure that the Window Manager creates for us. The data helps manage the window that will display the phrase "Hello world." `Rect` is a data structure that describes a rectangle object to QuickDraw. If you use the Metrowerks editor to examine the "Types.h" file, you'll find `Rect`, which looks like this:

```
struct Rect {
   short    top;
   short    left;
   short    bottom;
   short    right;
};

typedef struct Rect Rect;
```

`Top` and `left` correspond to the x and y coordinates of a point that QuickDraw uses in its drawing space. The `bottom` and `right` variables define a second point's coordinates. QuickDraw uses these two points to draw the rectangle. How does it make a rectangle made up of four points (or eight x and y coordinates) with just two points? QuickDraw relies on the fact that a rectangle can be drawn with this amount of data. First, QuickDraw draws a line from point (`top`, `left`) to point (`top`, `right`) to draw the top of the rectangle. Next, QuickDraw draws a line from point (`top`, `right`) to point (`bottom`, `right`), which draws the right side of the rectangle. Then QuickDraw follows with a line from point (`bottom`, `right`) to point (`bottom`, `left`) to draw the bottom of the rectangle. The line drawn from point (`bottom`, `left`) to point (`top`, `left`) closes the rectangle.

`MaxApplZone()` is a Memory Manager routine that ensures the application has sufficient memory. It does this by expanding the application's heap (also called a zone) as much as possible within the memory partition built for it by the Process Manager. If you don't call this routine, the Mac OS assumes a default heap size, which might not be adequate for your needs. This is followed by calls to `MoreMasters()`, a routine that allocates what are called master pointer blocks. These blocks contain pointers that help implement the handles that are frequently used to access Toolbox data structures. If you run out of master pointers, the Memory Manager will create more for you automatically. However, since the master blocks can't move about in memory, you run the risk of fragmenting the application's heap as memory

becomes littered with these immovable memory blocks. The application will also run more slowly as it struggles to organize the fragmented memory. If you provide sufficient master blocks now, it eliminates potential memory and performance problems in the future. Obviously, it's better to call `MoreMasters()` too much at initialization time, rather than too little.

Initializing Managers

Now we initialize the various Managers that we plan to use:

```
InitGraf(&qd.thePort);
InitFonts();
FlushEvents(everyEvent, 0);
InitWindows();
```

`InitGraf()` initializes QuickDraw. QuickDraw in turn sets up some global variables it uses to manage the application's graphic environment. The storage for these variables is set up by the development system, which QuickDraw accesses via the global pointer `thePort` that you provide. Next, the Font Manager is initialized so that text can be displayed within the window. `FlushEvents()` clears the event queues of any stray events when the application is launched. `InitWindows()`, of course, readies the Window Manager.

Now it's time to get into the actual mechanics of displaying the phrase "Hello world." Add to the program:

```
/* Set up the window */
   windowRect.top = windowRect.left = 40;
   windowRect.bottom = 200;
   windowRect.right = 300;
   if ((thisWindow = NewWindow(NIL, &windowRect,
      "\pHello world", IS_VISIBLE, documentProc,
      (WindowPtr) IN_FRONT, NO_CLOSE_BOX, NIL)) != NIL)
      {
      SetPort(thisWindow); /* Make window the current port */
      MoveTo (20, 20);
      DrawString("\pHello world");
      InitCursor();

      while (!Button()) /* Wait until mouse button clicked */
         ;
```

```
    DisposeWindow(thisWindow); /* Clean up */
       } /* end if */
    else
       SysBeep(30);

} /* end main() */
```

The first two lines of code plug coordinate data into the rectangle `windowRect` that are used to make the window. If you're puzzled over the point data's positive values, that's because in QuickDraw's coordinate system, the upper left corner of the screen is the origin, and larger positive numbers move a point toward the right and downward. The values in `windowRect` have QuickDraw create a window located forty pixels down and forty pixels to the right of the screen's origin. The window's upper left corner starts at this position, and the window is 200 pixels tall and 300 pixels wide.

The `NewWindow()` routine actually makes the window. The `#defines` we created at the top of the program are put to use here. From them we can surmise that the new window is visible on the screen, is supposed to appear in front of all other windows, has no close box (the small square in the window's upper left corner that, when clicked on, removes the window), and its title will be Hello World. `NewWindow()`'s first argument allows you to place a pointer to a data buffer for the window's use. If this argument is NIL, as it is in our example, the Window Manager allocates the window's data storage on the heap, which is fine for simple operations. However, if you display lots of text or large color images in the window, you can severely fragment the heap. For these jobs, it's best to pass the address of a memory block to `NewWindow()`. Consult *Inside Macintosh: Macintosh Toolbox Essentials* and *Inside Macintosh: Memory* for more information on these issues.

Notice that we do some error checking here. If `NewWindow()` successfully creates the window, it will return a pointer to the window's data structure. If `NewWindow()` has a problem making the window (possibly there's not enough memory), the routine returns a value of `NIL`. The `if` statement determines if we received a valid pointer from the Window Manager. If not, the application beeps and exits. Admittedly, a beep doesn't offer much diagnostic aid to the user, but it's preferable to signal a problem this way and quit cleanly, rather than have the Mac crash.

If we have a valid window pointer, the program next sets the window to be the current drawing port by using `SetPort()`. QuickDraw always draws to the

screen through a graphics port or *grafport*, which is another data structure that describes to QuickDraw an area to draw on the screen, the size and shape of this area, its coordinate system (which can be different from the screen's), what type of text to use, and other information. The Window Manager creates a grafport for every window it makes, and your application can create and manage many windows—and thus grafports—at once. Through the `SetPort()` routine, we inform QuickDraw what grafport to draw in, which in this case is our shiny new window. The following `MoveTo` routine nudges the current drawing point within the window down and right twenty pixels. These values use the window's own coordinate system, whose origin is located at the window's upper left corner. Finally, we use the `DrawString()` routine to write the phrase "Hello world" in the window.

When the Process Manager starts the application, it changes the mouse pointer, or cursor, to a stopwatch to indicate the Mac is busy. Now that our initialization code has completed and the program displays the greeting, we call `InitCursor()`, which changes the cursor back to an arrow. This indicates that our application is ready to deal with the user.

If we simply let the program proceed, the window would appear briefly and be gone. To let the window linger so that we can admire our handiwork, we insert a `while` loop. This loop cycles until the routine `Button()` returns `TRUE`, which occurs when you press the mouse button. Once the loop completes, we clean up after ourselves by calling `DisposeWindow()`, which removes the window and purges the data structure made by `NewWindow()`. The final shape of the program looks like so:

```
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Windows.h>
#include <Memory.h>
#include <Events.h>
#include <OSUtils.h>

#define NIL          0L
#define IN_FRONT     (-1)
#define IS_VISIBLE   TRUE
```

```
#define NO_CLOSE_BOX FALSE

void main(void)
{
WindowPtr   thisWindow;
Rect        windowRect;

/* Lunge after all the memory we can get */
   MaxApplZone();
   MoreMasters();
   MoreMasters();

/* Initialize the various Managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();

/* Set up the window */
   windowRect.top = windowRect.left = 40;
   windowRect.bottom = 200;
   windowRect.right = 300;
   if ((thisWindow = NewWindow(NIL, &windowRect,
      "\pHello world", IS_VISIBLE, documentProc,
      (WindowPtr) IN_FRONT, NO_CLOSE_BOX, NIL)) != NIL)
      {
      SetPort(thisWindow); /* Make window current drawing port */
      MoveTo (20, 20);
      DrawString("\pHello world");
      InitCursor();

      while (!Button())    /* Wait until mouse button clicked */
         ;
```

```
        DisposeWindow(thisWindow);
        } /* end if */
    else
        SysBeep(30);
} /* end main() */
```

Run the Code

Let's compile and run this code. Using the seven-step procedure outlined in Chapter 2, first save the code (if you typed it) into a file called `Hello1.c`. Next, create a project called `Hello.μ`. Add "Hello1.c" to it, and remove the placeholders from the project's window. (You don't need to do this with CodeWarrior Lite on the Power PC Programmer's Toolkit CD, because the project file is already set up.)

Set the preferences in this project for the C/C++ Language, and PPC Project panels. For the Language preferences panel, ensure that the Require Function Prototypes item is checkboxed and that the Prefix File item is blank. In the C/C++ Warnings panel, see that the usual items are checkboxed: Unused Variable, Unused Arguments, and Extended Error Checking. For the PPC Project preferences panel, name the output file `Hello`. Now make the project and run it. You'll get a window that resembles that shown in Figure 3.1.

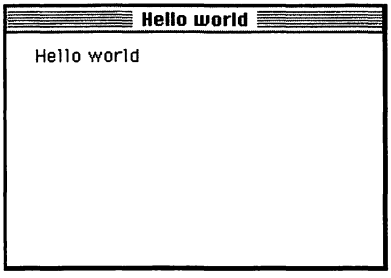


Figure 3.1 The result of the "Hello world" program.

Click on the mouse button to quit the application. The font used in the window was the default application font Geneva. One of Mac OS's finer features is that it has a smart set of defaults, which simplifies programming.

It took 50 lines of code to implement the "Hello world" program. Our resulting application doesn't do much—but then neither does the Unix-style version of the program that every beginning C programmer writes. It does illustrate that the Mac OS is a complex environment that requires attention to a lot of details before you can write code.

This very simple application required that we have a grasp of the Memory Manager, the Window Manager, and QuickDraw. I've only provided superficial descriptions of some of the Toolbox routines used in the setup code. For additional information, consult *Inside Macintosh: Macintosh Toolbox Essentials, Inside Macintosh: Memory,* and *Inside Macintosh: Imaging.*

"Hello1.c" demonstrates the general initialization setup for a Mac application. Later programs will require the setup of more Managers, but these will just be additions to the code you've written here. Like the understanding of the Mac itself, Mac programming is just a matter of continually adding components to a basic structure.

The Fork in the File

Now that we've covered program initialization, let's delve into a Mac file's structure. A Macintosh file is composed of two sections, a *data fork* and a *resource fork.* Physically, there's nothing different about these forks; each is simply a stream of bytes located somewhere on the hard disk. However, the Mac OS treats each file fork differently. The data fork typically contains data created by an application, such as text from a word processor, numbers from a spreadsheet, or PostScript commands from a drawing application.

The resource fork is a container for objects called—you guessed it—resources. Resources contain data that is organized into predefined formats. This data typically describes graphic elements such as icons, windows, and color tables. Resources also contain nongraphic yet essential elements such as drivers or program code. A resource type defines the resource to the Mac OS so that it can properly interpret the data packaged within the resource. A resource type is a four-character code, such as 'CODE', 'MENU', 'WIND', 'cicn', 'cdev', and so on. As examples of how the resource type indicates what is inside a resource, consider that CODE resources contain processor code, MENU resources contain the items that appear on a menu, and cicn resources hold data that displays a color icon. In summary, the resource fork of a 680x0

application contains such elements as program code, menu lists, windows, and icons. The structure of a Power Mac application is somewhat different: It still keeps the graphical elements in its resource fork, but the program code is stored as a single block inside the file's data fork. More on this later in Chapter 4. For more details on a file's data and resource forks, consult *Inside Macintosh: Files*, and for more on resources, check *Inside Macintosh: Macintosh Toolbox Essentials.*

Future Directions

Copland will use a completely re-engineered version of the File Manager to handle larger volumes, and it will use more efficient I/O algorithms to improve performance and reliability. One feature of this new File Manager is that it supports any number of file forks, besides the usual data and resource fork. The structure and contents of these additional forks are determined by the programmer. For example, a game designer might store the bitmaps for sprites in one custom fork, and the texture maps for objects and hallways in another.

Besides the two forks, each file also has a type and creator. Like resource types, file type information is a four-character code that describes a file's contents to the application that opens it. For example, a file type of 'TEXT' indicates that the file contains ASCII text, 'TIFF' indicates the file has Tagged Image File Format bit-mapped data (typically a scanned image), and 'APPL' means the file contains program code organized as an application. The creator information is a four-character code signature that's unique to the application that created the file. Each file's type and creator information is maintained in a desktop database file by the Mac OS. The Finder, the shell application that displays and manages the so-called virtual desktop on your Mac's screen, uses the database file to display each file's icon at the appropriate screen location. Where does the desktop database get a file's type and creator information, along with its file icon? From resources in your program, of course.

To see how all this fits together, consider what happens when you double-click on an document icon (say, a CodeWarrior project file). The Finder detects this action, and obtains the file's creator information from the

desktop database. Next, it searches for a file of type 'APPL' (an application) with the same creator signature. If the Finder finds this application file (the CodeWarrior IDE), it has the Process Manager launch it. If the Finder can't locate the application file, you get a warning on-screen that states: "The document 'Foobar' could not be opened, because the application program that created it could not be found."

Obviously, the Metrowerks CodeWarrior IDE manages the CODE resources in the application that we make. However, to build a complete Mac application with menus, windows, its own custom icon, and signature information, it's probably dawning on you that you're going to have to become familiar with resources in greater detail. This assessment is correct, so let us begin.

Making Resources

As usual, the best way to learn about resources is to do something with them. A great place to start would be to put a friendly interface on that user-hostile file munger program we wrote in Chapter 2. First, consider what we want the munger program's interface to do. It should basically behave as before and let you pick a file to open, ask you to name an output file, and then process the chosen file. When munger finishes the job, you want a status report. Once you've finished processing one or more files, you quit munger. With some thought, we conclude that all the munger application really needs is an Apple menu, a File menu, and an Edit menu. The Apple menu is just a placeholder for an application's About Box, the window where the program's description hangs out. The File menu needs an Open command to open the desired files and a Quit command to exit the program. The Edit menu won't be of much use to our application; it's there to assist passing events to other applications under System 7's cooperative multitasking environment. We also need to design dialog boxes, which are the windows that display processing statistics and warn of problems. Finally, we want to display a cool About Box dialog box that describes munger when the About command is chosen from the Apple menu.

Locate ResEdit, the resource editor, in the Apple Tools folder on the CodeWarrior CD-ROM and copy it to your hard disk, if you haven't done so already. ResEdit lets you create resources, modify them, and save them to a file's resource fork, much like a text editor does with text data in a file's data

fork. Launch ResEdit. Click on the splash screen to dismiss it. Click on the New button. When the Standard File dialog box appears, type `munger.π.rsrc`. (Remember, to get the π character, type Option-P.)

Hazard

Previous versions of the CodeWarrior IDE required that the resource filename you chose closely matched the project's filename. That's because when you test drive an application in the CodeWarrior IDE, it does some important housekeeping for you. By default, it searches for resources (except for the CODE resources that it made) in a separate file whose name begins with the project name and ends with the string ".rsrc." For example, for project munger.μ, we could keep our resources in a file called "munger.μ.rsrc." This arrangement lets you rapidly modify graphical resources in the resource file without having to attach them to the program's resource fork every time you want to test changes to the interface.

Starting with version 1.3 of the CodeWarrior IDE, the naming convention of the resource file has been relaxed. That's because you can add a resource file to the project, just like source code files and library files. Just go to the Project Menu and select the Add Files... item. Then use the Standard File dialog to locate and pick any resource files. Their names will appear in the project window. Now when you build the program, the CodeWarrior linker automatically appends any resources these files contain to the resulting application file. If you wish, you can still use the naming convention and let the CodeWarrior IDE manage locating and linking the resources for you. For this example, we'll fly in the face of convention and give the resource file a name different from the project file's to demonstrate how to add the resource file to the project.

A window called munger.π.rsrc appears. This window serves as a view of the file's resource fork. It's empty because there are no resources in it—yet. Thinking back to our interface design meeting a little while ago, we decided that munger needed several menus. Go to the Resource menu and choose the Create New Resource command, as shown in Figure 3.2.

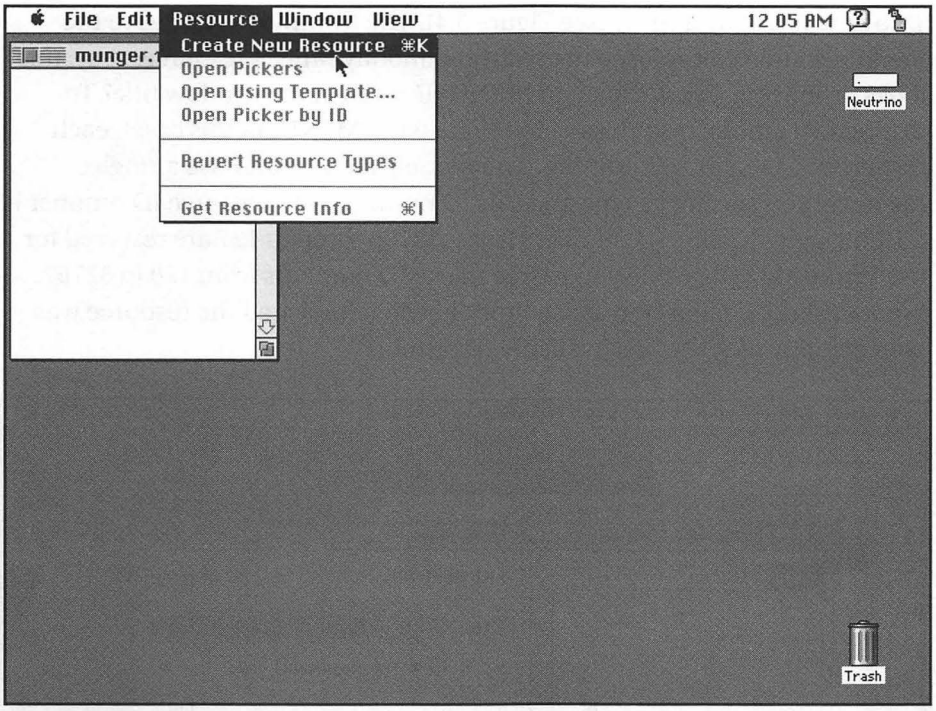


Figure 3.2 Preparing to make a new resource in ResEdit.

Making Menus

A dialog box appears, asking for a resource type. You can either scroll through the list of defined resource types or type in one if you know it. Type MENU (as shown in Figure 3.3) and press Return.

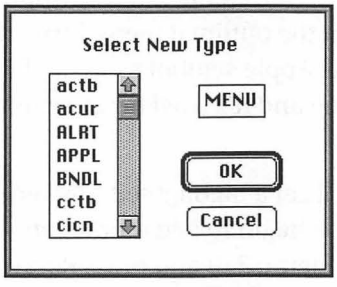


Figure 3.3 Making a MENU resource.

Two new windows appear (see Figure 3.4). The frontmost belongs to the menu resource editor, used to create and modify MENU resources. Say, this looks promising. But what's that MENU ID = 128 in the window title? To distinguish among resources of the same type (MENU, in this case), each resource has its own ID number. To uniquely identify and use a single resource, you specify its type and this ID number. The resource ID number is a 16-bit signed value. ID numbers from -32768 through 127 are reserved for use by the Mac OS, while you're free to use ID numbers from 128 to 32767. What ResEdit's menu resource editor did when it created the resource was conveniently pick the first available ID number.

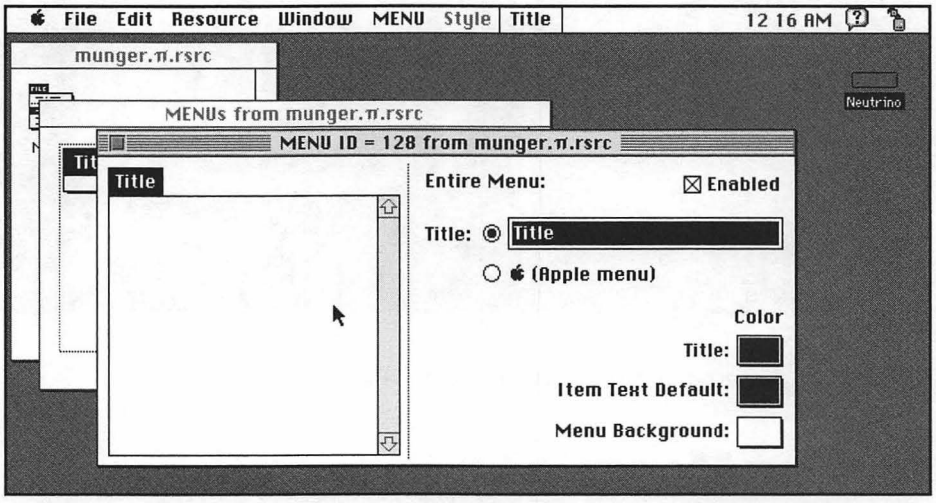


Figure 3.4 The MENU resource editor.

Since the first menu is the Apple menu, click on the Apple menu radio button in this window. The word Title changes to the Apple symbol, as shown in Figure 3.5. Note also that the outlined menu formerly named Title in the menu bar changed to the Apple symbol as well. This menu is a clone of the menu you're constructing and it's used for examining a menu's arrangement and appearance.

Now, press Return. You'll get a highlighted (darkened) area under the Apple symbol. This is where you begin to add menu items. For the Apple menu, type **`About Munger...`** (see Figure 3.6) and press Return. This is the program's About Box menu item.

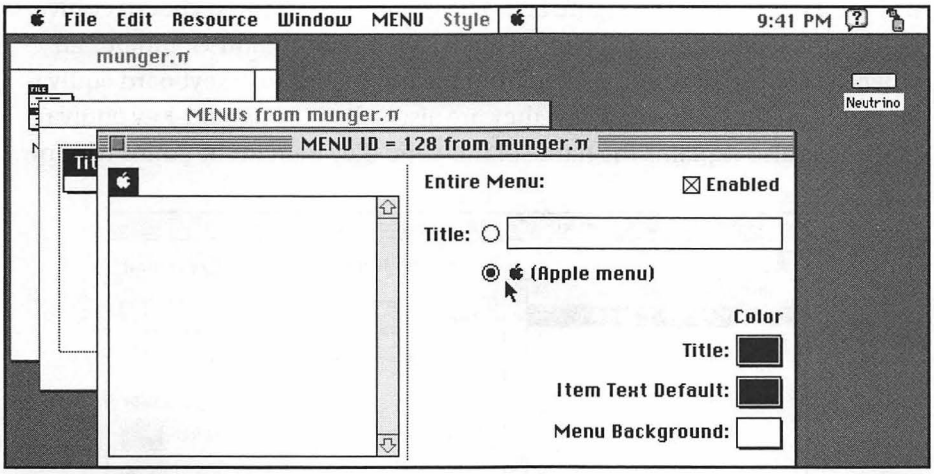


Figure 3.5 Making the Apple menu.

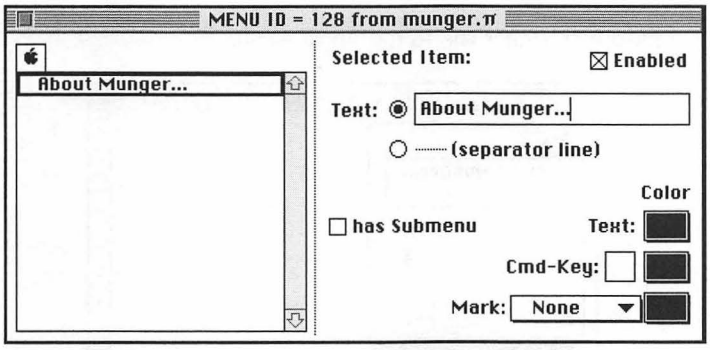


Figure 3.6 Making the About Box menu item for the Apple menu.

For the Apple menu, the next menu item is simply a separator or divider line, used to indicate where the application's menu ends and the rest of the Apple menu begins. To add a separator line, click on the separator line radio button, as shown in Figure 3.7.

Now click the window's close box and you'll see MENU resource 128 (see Figure 3.8).

We still have two more menus to go. Once again select Create New Resource from the Resource menu, or type Command-K. A new MENU ID = 129 window appears. Enter `File` for the menu's title, press Return, and type `Open...` for the first menu item. Before you press Return, click on the box to the right of the item labeled Cmd-Key in the Editor window, or press Tab to select

it. Type **o** in this box (see Figure 3.9). The O character is the keyboard equivalent for the Open menu selection. That is, typing Command-O initiates an Open action, as if it were selected from the menu. Because keyboard equivalents rely on the Command key, they are also called Command-Key equivalents. This also explains the name of this Cmd-Key item in the editor window.

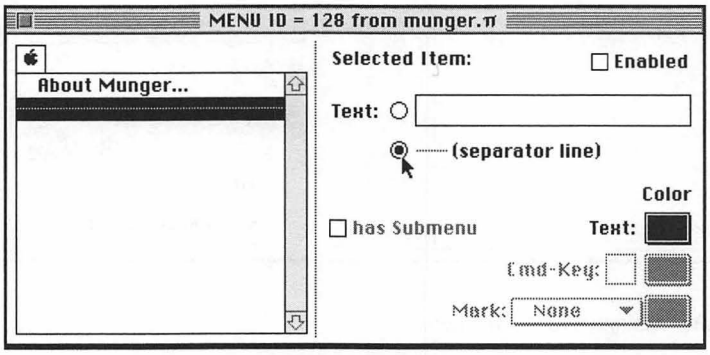


Figure 3.7 Adding a separator line to the Apple menu.

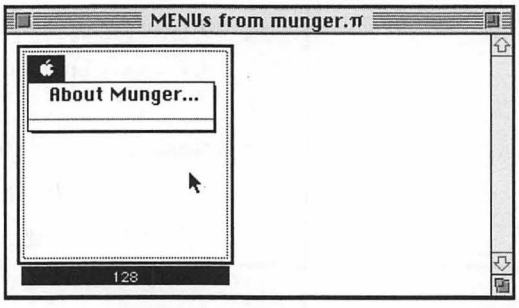


Figure 3.8 MENU ID 128, as it will appear in the application.

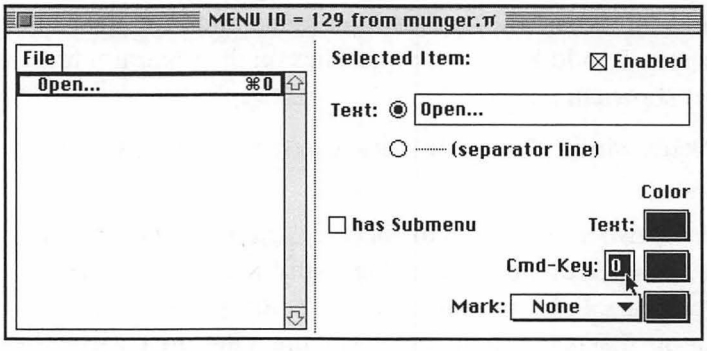


Figure 3.9 Entering the keyboard equivalent for the Open menu item.

Press Return and then add a separator line by clicking on the separator radio button. Press Return again and type `Quit`. Then, type a `Q` in the Cmd-Key item box. That completes the File menu. You can then pull down the test menu to examine it (see Figure 3.10). Click on the window's close box and save the file.

Now to add the last menu, the Edit menu. Type Command-K to create a new menu resource. The window MENU ID = 130 appears. Type `Edit` for the menu title, press Return, type `Undo`, press Tab, type `Z`, and press Return, which makes the Undo item in the menu. It has the keyboard equivalent of Command-Z. Add a separator line and press Return, type `Cut`, press Tab, type `X`, then press Return to add the Cut item to the Edit menu. Add the Copy item by typing `Copy`, Tab, `C`, and pressing Return, then type in `Paste`, Tab, and `V` to create the Paste item. Click the window's close box, and you should see all three menus, ready to go, as shown in Figure 3.11. Save the file, and close the window by clicking on the close box, or typing Command-W.

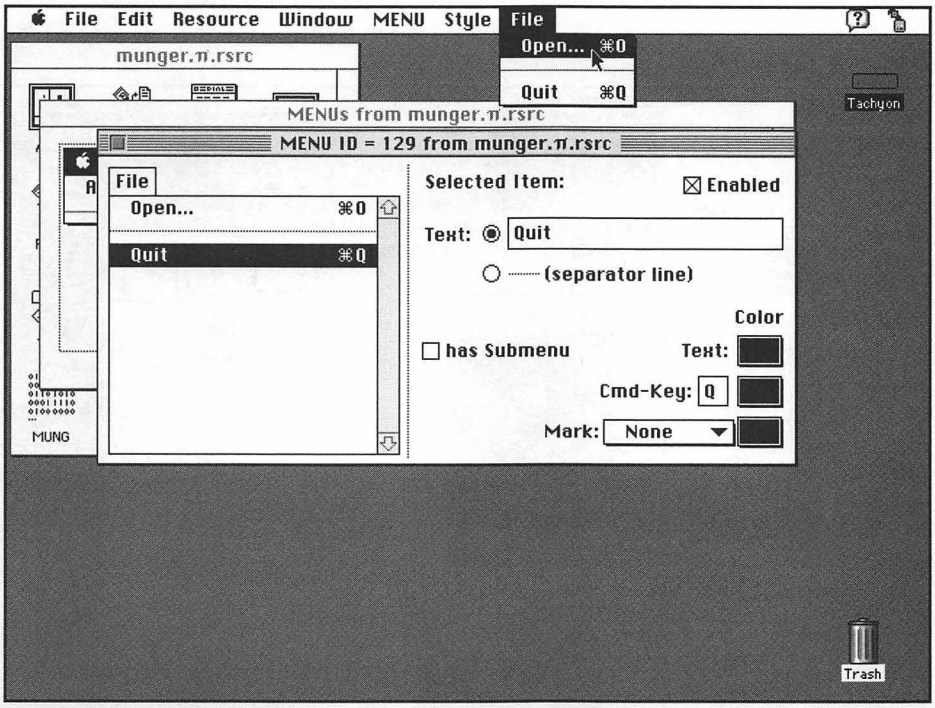


Figure 3.10 Testing the completed File menu in ResEdit.

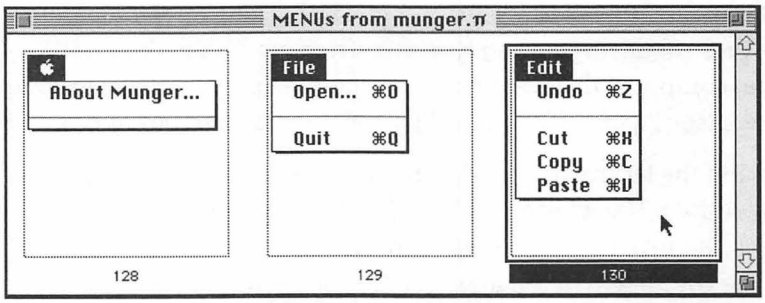


Figure 3.11 The complete MENU IDs for the munger application.

Making Dialog Boxes

Now, let's make the dialog boxes for munger. Choose Create New Resource again, and this time type `DLOG` and press Return. A dialog editor window opens, with the title DLOG ID = 128 (see Figure 3.12).

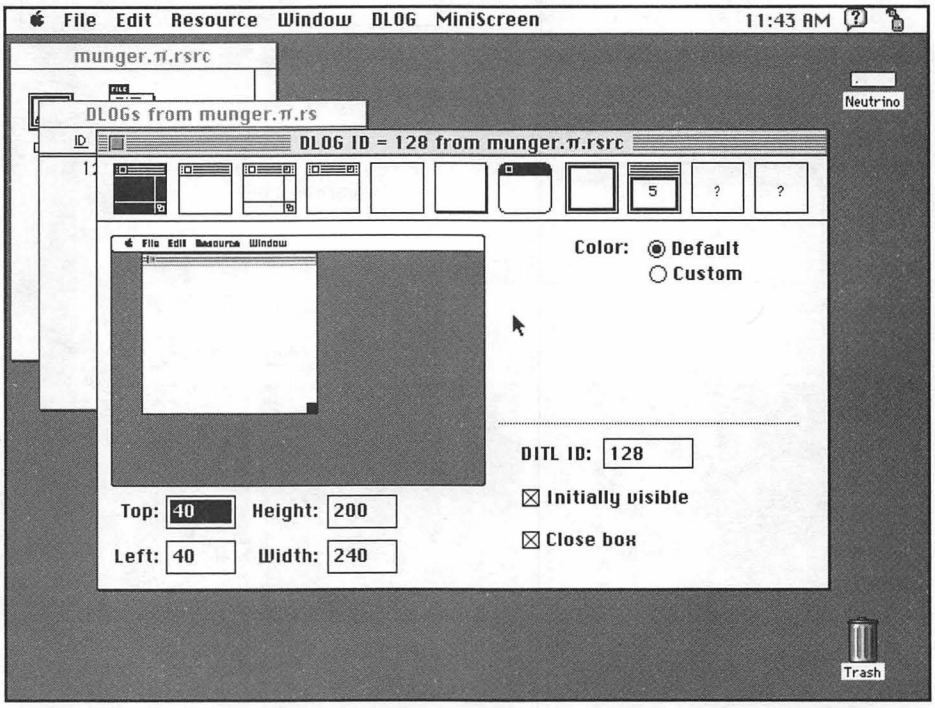


Figure 3.12 The dialog editor window.

About Boxes are typically dialog windows, because this type of window requires little program code to support it. By default, the editor has selected a standard document window, complete with a drag region, a close box, and a grow icon (the small box at the window's bottom right corner). In short, a window with all the bells and whistles. Go over to the sixth window icon from the left and click on it, as shown in Figure 3.13. Notice that the window's appearance has changed. This is the alternate dialog window, which is just a variation of the dialog window. This window type has no drag bar, no close box, and no grow icon. It's pretty simple as windows go, which is what we want.

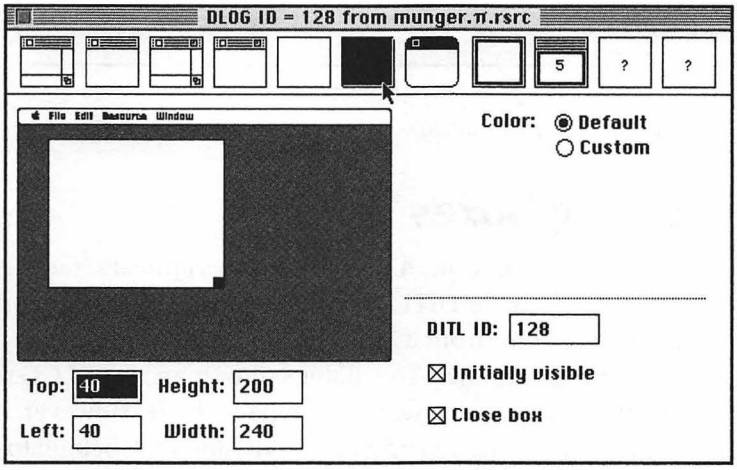


Figure 3.13 Picking the alternate dialog window.

Click on this window's upper left corner and drag it near the top of the screen. Next, click on the dark square at the bottom right of the window, and drag it. The window's size will change depending upon how you drag this square. Size the window according to what suits you, and release the mouse button (see Figure 3.14).

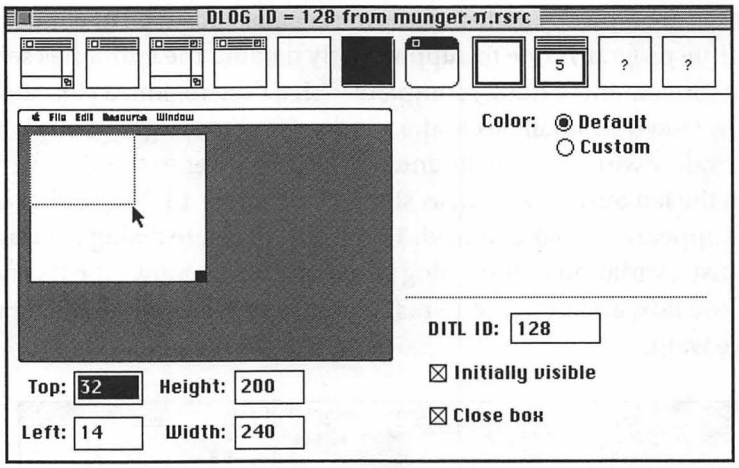


Figure 3.14 Resizing the dialog window.

Editing Dialog Boxes

Now, double-click on this window. A pair of windows appears (see Figure 3.15). This is the dialog item, or DITL, resource editor. While the menu editor lets you add and delete items from a MENU resource, the situation is more complicated with dialog windows. The dialog editor manages DLOG resources, which determine a dialog window's type and size. However, objects that appear in the window, such as buttons, icons, and text, belong to another resource, of type 'DITL'. DITL resources contain lists of dialog items, just as MENU resources contain lists of menu items. Naturally, changing DITL resources requires a separate editor, which is why that dialog item editor just appeared.

Although the DLOG and DITL editors operate so seamlessly that they appear to function as a single editor, it's very important that you remember that you're working with two different resources here. Notice that the DITL ID number is 128. It's not required that a dialog's items (DITL resources) have the same ID number as the dialog window (DLOG) that they appear in, but it does keep tracking the relationships between the two resources simple. If you need to use a different DITL ID number, you can change the linkage by typing a different ID number in the DITL ID item on the DLOG Editor window in the background.

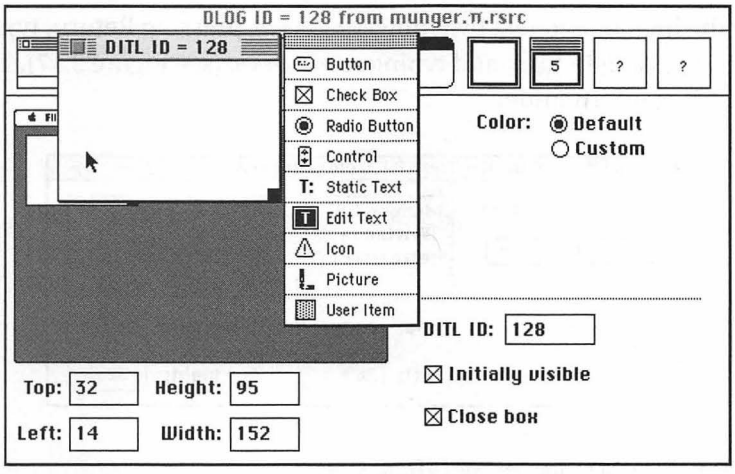


Figure 3.15 The DITL editor, for modifying dialog items.

Go to the floating window with the dialog items on it (the window at the right), and drag the static text object to the dialog window, as shown in Figure 3.16. Static text can't be changed by the user during the life of the dialog window, so it's useful for handling the titles of buttons and controls.

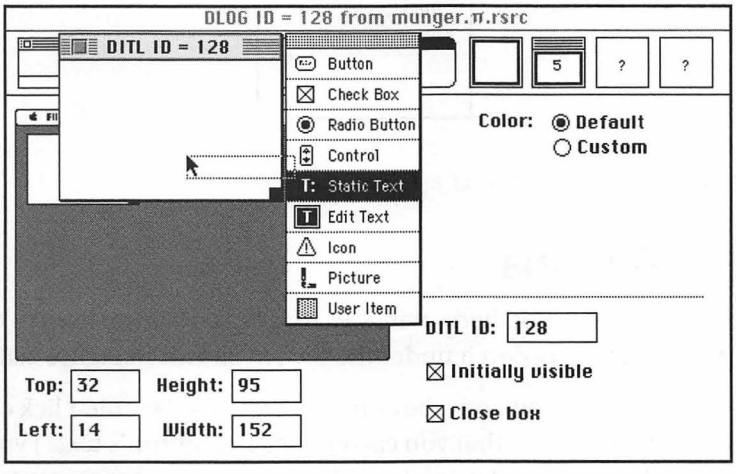


Figure 3.16 Adding a static text item to the About Box.

Release the mouse button when you've positioned the text object where you want it. In this example, let's drop it near the top of the window. Now double-click on this object, and a window titled Edit DITL item #1 appears.

Replace the highlighted text by typing `Munger 1.0`, pressing Return, typing `Written in`, pressing Return, and typing `Metrowerks C` (see Figure 3.17). This is our About Box information.

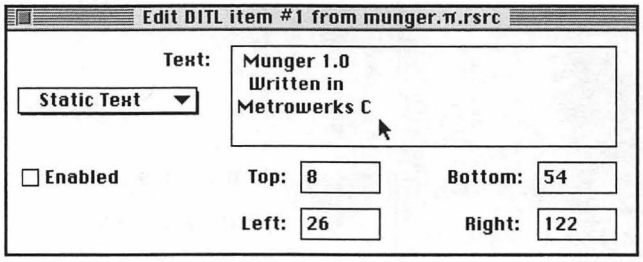


Figure 3.17 Changing the text of DITL item #1.

Click on this window's close box, and resize the static text box by clicking and dragging with the mouse (see Figure 3.18). You'll have to tinker with the box and text somewhat until you get it to look neat. Use ResEdit's Alignment menu to center this text box in the window.

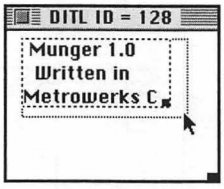


Figure 3.18 Modifying the size of the dialog item.

Adding Buttons

Now, go back to the dialog items window, and drag a button item to the dialog window, and position it under the text, as shown in Figure 3.19.

Release the mouse button and a button item appears. Double-click on it to open an Editor window so that you can change the button's text. Type `OK` (see Figure 3.20). Close the window and use the Alignment menu to center the button.

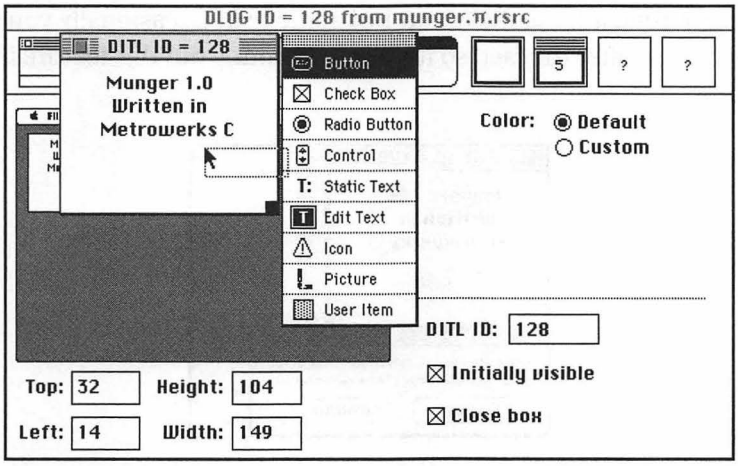


Figure 3.19 Adding a button to the dialog window.

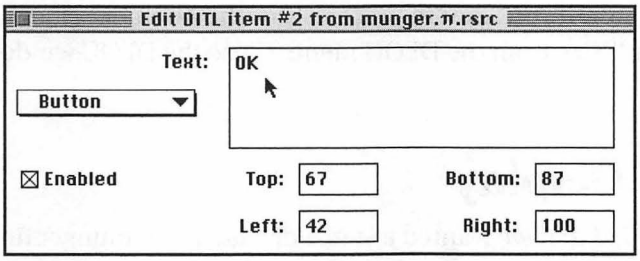


Figure 3.20 Changing the button's title.

Numbering Dialog Items

There's one more crucial step we have to do here: renumber the dialog items. The reason is that certain dialog Toolbox routines that manage the dialog items look for Return keystrokes. They pass this action onto the first item in the dialog list, just as if you had clicked on that item. What we want to happen is that when the user presses Return, it activates the OK button, which then dismisses the About Box window.

Go to ResEdit's DITL menu, and select Renumber Items.... A new window appears, with instructions on how to renumber the items. Hold down the Shift key, then click first on the OK button, and then the About Box information (see Figure 3.21). Click on the renumber button, and you're done. You could have avoided renumbering these items by putting the OK button in the

window first, then adding the About Box static text. Occasionally you have to renumber items after the fact, so it's worth pointing out this feature in ResEdit now.

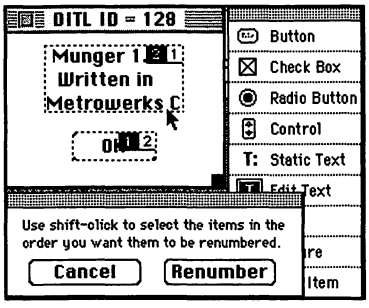


Figure 3.21 Changing the dialog item numbers.

Close the DITL editor by clicking on the close box, which lands you back in the DLOG editor. If you want to preview how the About Box looks, pick Preview at Full Size from the DLOG menu. Close the DLOG window and save the file.

Status Display

We also decided that we wanted a status display when munger finishes processing a file. Let's start by typing Command-K to create a new DITL resource. As the title to the DITL Editor window indicates, this resource has an ID of 129. Click on the eighth window from the left to select the dialog window type. The window changes from a document window type to a dialog window, as shown in Figure 3.22.

Double-click on the window to bring up the DITL resource editor. Go to the floating dialog item window and drag a static text item to the new window. Adjust the item's width by dragging with the mouse until the item spans most of the window. Now copy and paste this item. Nothing appears to have happened, but if you click and drag on the static text item, you'll uncover an identical static text item beneath it. Copy and paste again to clone the item one more time, then arrange the three items above one another in the dialog window. This gives you three static text items of the same size. Use the Alignment menu to center the items in this window, as shown in Figure 3.23.

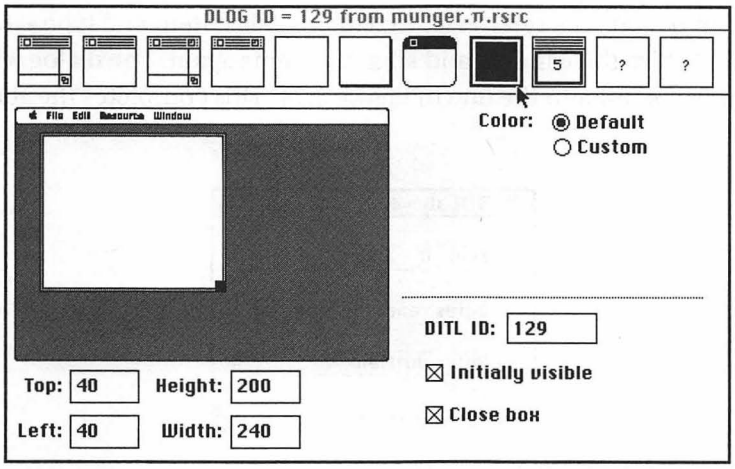


Figure 3.22 Changing the window type to a dialog window.

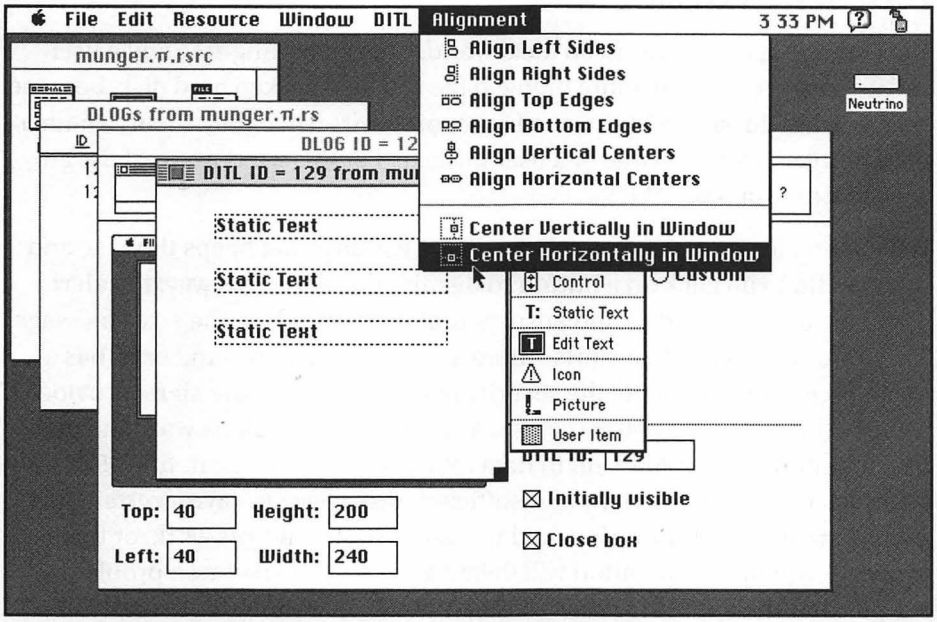


Figure 3.23 Centering the three cloned dialog items.

Click the top static text item to select it and edit its contents by double-clicking on it. Type `File: ^0`. The caret and number operate as a special placeholder where the dialog Toolbox call will substitute a text string, in this case a filename. We'll see how this works a little later. Go to the second item,

open the item, and type `Bytes read: ^1`. Open the last item and type `Bytes written: ^2`. Resize the window and align the items again. The dialog window should appear similar to the one in Figure 3.24. This completes the status window.

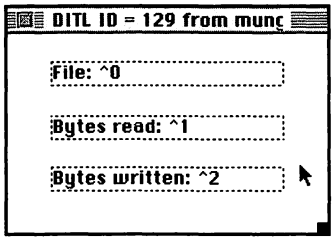


Figure 3.24 The completed status dialog box.

Adding Alerts

Now for one last window. In an ideal world, our code is bug-free and a user will never try to add one more munged file to a jam-packed hard disk. Because such a world doesn't exist, we need to report errors when they occur, whether it's a problem with the code or a user mistake. For this window, we'll use an alert resource of type 'ALRT'.

What is an alert? An alert is a special dialog window that beeps the Mac and requires that you click on a button to dismiss the alert. This way, the alert grabs the user's attention and ensures that he responds to the error message. There are several types of alerts—note, caution, and stop—and each has a distinctive icon to indicate the severity of the problem. Note alerts provide information, usually to offer the user a choice. Caution alerts warn the user of a situation that could result in data loss if not dealt with carefully. For example, caution alerts warn of insufficient disk space to save a certain file, or that memory is running low and the user should save his work, or that proceeding with an operation will delete a file. Stop alerts flag a problem so serious that the application can't complete the operation. An example of a stop alert is when the program detects an error while writing a file to disk.

For the munger program, we can anticipate that disk I/O is where most problems will occur. Since most disk I/O problems—such as running out of disk space—are difficult to recover from without lots of intervention on the user's part, munger will just quit the operation and post a stop alert.

Let's make a stop alert for munger. Get out of ResEdit's DLOG editor by closing the Editor window. Type Command-K to make a new resource, and type ALRT in the Select New Type dialog. The alert resource Editor window appears, with a default of ALRT resource ID 128. Notice that the alert window already has dialog items in it (see Figure 3.25). Remember that the objects displayed in the dialog box actually belonged to a different resource? What's happening is that the alert editor is, by default, using DITL resource ID 128, whose items already belong to the About Box dialog.

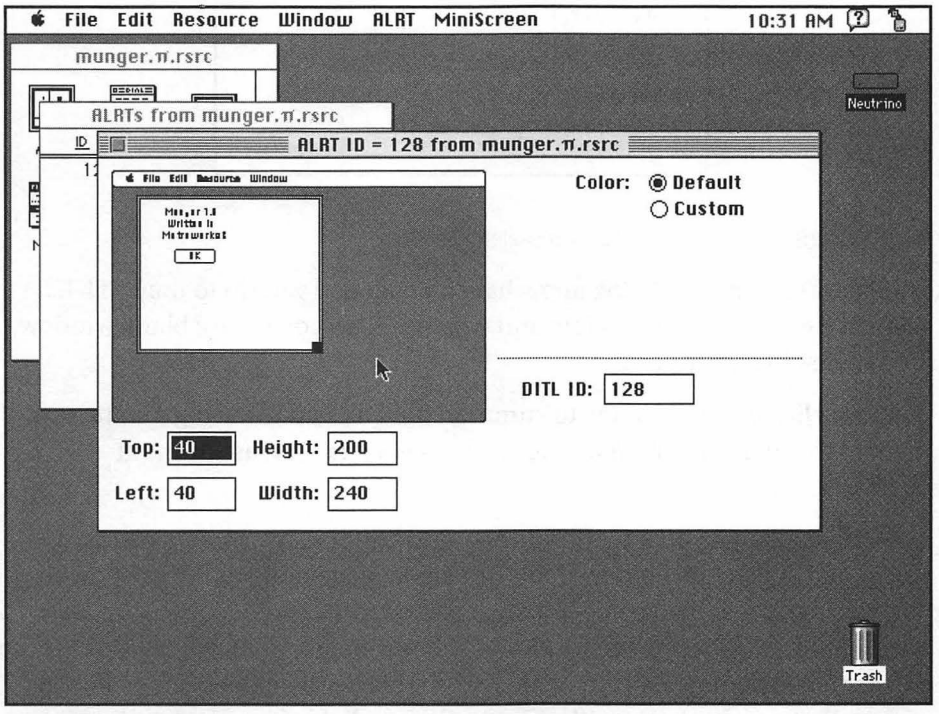


Figure 3.25 The alert resource editor.

You have two options here. You can change the ID number that links the DITL resource to the ALRT resource, or change the ALRT resource ID. I keep the organization of these linked resource ID numbers simple by using an ascending list of ID numbers that's divvied up among the DLOG and ALRT resources. That is, an ALRT resource might get an ID of 128, a dialog a resource ID of 129, another ALRT gets ID 130, and so on. So, let's change the

ALRT resource ID. Start by selecting Get Resource Info from the Resource menu, or typing Command-I. An Info box appears (see Figure 3.26). Type `130` to change the ALRT ID, and then close the window.

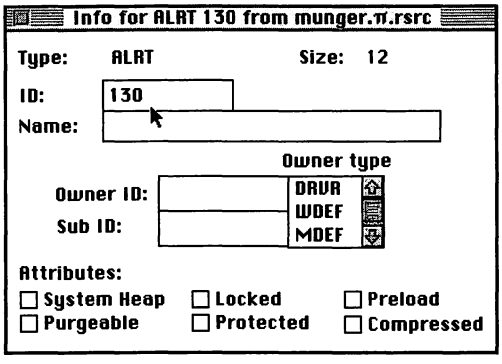


Figure 3.26 Changing the ID of the alert resource.

You'll notice that the dialog items haven't changed yet. Go to the DITL ID item in the alert Editor window and type `130`. Now you have a blank window, as appears in Figure 3.27.

Double-click on the window to summon the DITL editor. Drag a static text item to the window, and edit it to say `I/O error, ID = ^0`, as shown in Figure 3.28.

Now drag a button item to the window and edit it to say `OK`. Align the two items and resize the window to fit. Be sure to leave room at the window's top left corner so that the Dialog Manager can drop a 32-by 32-pixel stop alert icon into the window when it's drawn. Renumber the dialog items so that the OK button is item number 1. Again, we do this because the Dialog Manager passes Return key events to the window's first dialog item, and we want that to be the OK button. Also, for alerts, the Dialog Manager draws a bold outline around DITL item 1, on the assumption that it's the default button (an OK button in this instance). The alert window should appear as shown in Figure 3.29.

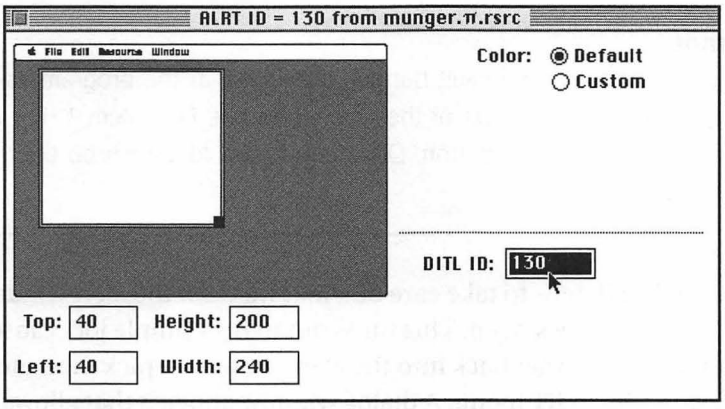


Figure 3.27 Changing the ID of the alert's DITL resource.

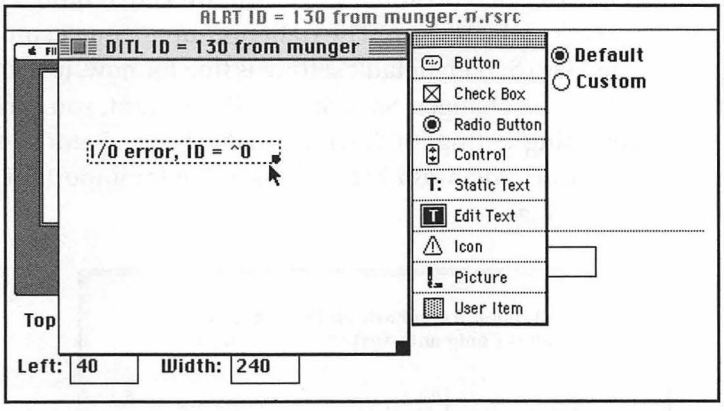


Figure 3.28 Adding the alert's static text message.

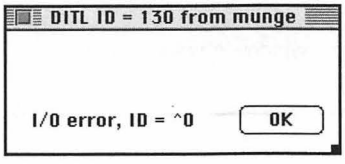


Figure 3.29 The completed alert dialog.

Important
As discussed earlier, the default button—the item that the program assumes the user will pick most of the time—is always DITL item 1. For alert boxes with more than one button, DITL item 2 should always be the Cancel button.

There are two last details to take care of. First, we want the alert window to appear centered on the screen. This turns out to be a simple job. Close the DITL Editor window to get back into the alert editor and pick the Auto Position... from the ALRT menu. A dialog window appears that allows you to set the window's characteristics so that System 7 will automatically center the window for you (see Figure 3.30). Go to the active pop-up menu (the one on the left), and select the alert position. (Alert windows are required to appear on certain areas of the screen.) The right pop-up menu becomes active, but since the Main Screen default setting is fine for now, just click on the OK button to make the changes. Save the file. If you want, you can also enable the auto-centering settings of the other dialog boxes. Before you do, consult *Inside Macintosh: Macintosh Toolbox Essentials* for important guidelines on these settings.

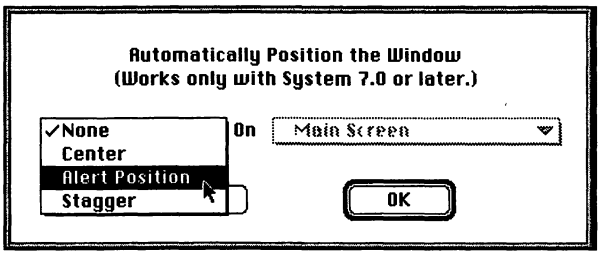


Figure 3.30 Setting the alert window's screen position.

Background Info
In pre-System 7 days, dialog boxes would appear on-screen where they were drawn in the resource editor. Because monitors of any size and shape could be attached to the Mac, the default location of these windows weren't always in the best position for visibility, especially on a large

monitor. You could always write code to determine the Mac's screen size and then position the dialog window appropriately before showing it. This code could get extremely complicated if the system had multiple monitors in use. While such code isn't impossible to write, it was an imposition on the programmer's resources, which were better spent writing the application, not managing the interface. As you saw with the alert editor, System 7 now handles this job. This is one of the many improvements in System 7 that both relieves the programmer of an interface detail, and makes applications more visually consistent to the user.

The other detail is that the dialog item lists aren't cleared from memory automatically when the alert or dialog box is closed. To help the Memory Manager reclaim the memory used by these item lists, we mark the DITL resources as purgeable. To do this, first close the alert resource editor window and then the ALRT resource window. The resource fork of "munger.π.rsrc" should contain four resources, as shown in Figure 3.31.

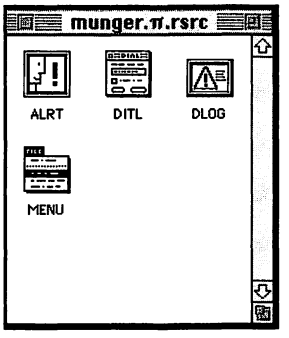


Figure 3.31 The resources the munger application uses.

Next, double-click on the DITL icon to get a view of the DITL resources, from 128 to 130. Hold down the Shift key and click on each DITL resource to select it. Choose Get Resource Info from the Resource menu and three Info windows should appear, as shown in Figure 3.32. Click on the Purgeable checkbox for each DITL resource to select Purgeable. Close the windows. Save the file and quit ResEdit.

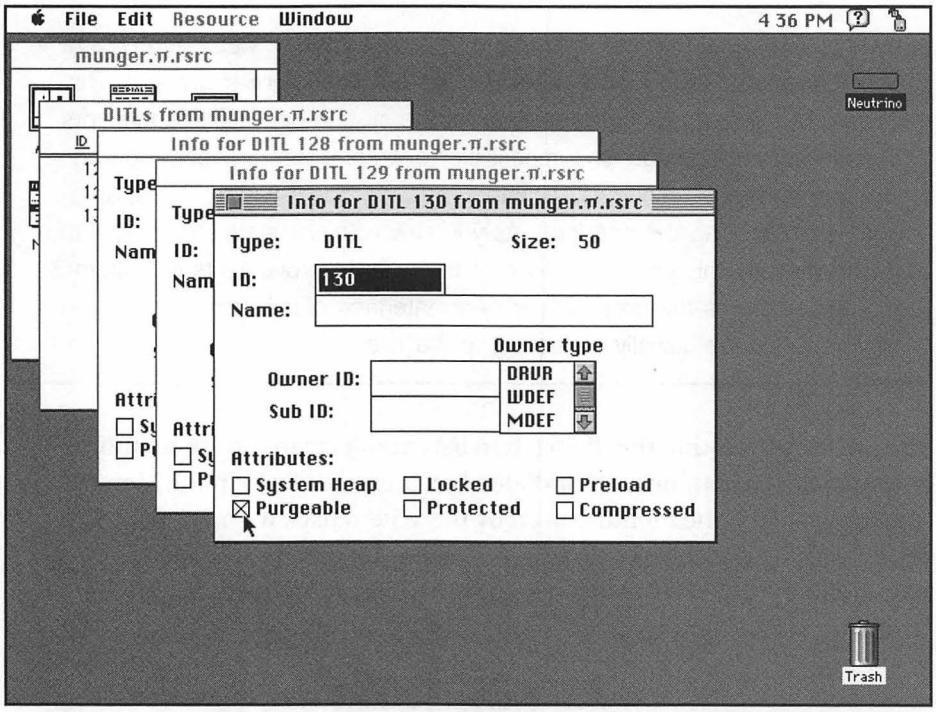


Figure 3.32 Setting the DITL resources as purgeable.

Saving Resource Data as Text

This excursion with ResEdit covered a lot on resources. You might be wondering if there is a way to save the information they represent in a text format. This would allow a resource's contents to be distributed on paper, or as 7-bit ASCII over the Internet. For huge programs with dozens of menus, windows, dialogs, and alerts, it's easier to search for an item to modify using a text editor rather than poking around in a resource fork with ResEdit. And yes, you can save the information in a text format. Along with ResEdit, CodeWarrior supplies two Macintosh Programmer's Workshop (MPW) tools, called DeRez and Rez. The DeRez tool takes an existing .rsrc file and translates its resources into text descriptions. The text description uses a C-style programming language that accurately describes a resource's data. The Rez tool takes these text files and converts them back into binary resources. A new version of Rez now "plugs" into the CodeWarrior IDE so that you don't have to deal with the MPW environment at all. You can access the MPW

version of these tools from the Metrowerks CodeWarrior IDE by selecting Start Toolserver from the Tools menu. The plug-in version of Rez simply requires that you add the text description file to your project. For more information on how to use Rez and the Toolserver, see Chapter 5.

Some Words on Events

Now that we've got the new and improved munger interface constructed, we're almost ready to start writing code. First, a brief description of how a Mac application operates is in order. As you work with the Mac, you generate events. There are two types of events: low-level and high-level. Low-level events are actions such as keystrokes, mouse clicks, and the insertion of the occasional floppy disk. The Mac OS uses the Event Manager to detect these actions and place them in an event queue for the application. High-level events are used to establish communications among applications. Such communications might request data from another application, or command an application to print a file. We'll deal with high-level events later in this chapter.

Your application takes these events from the queue and responds to each type as required. It does this using what's called an event loop. In the event loop, the application circles endlessly, obtaining events from the OS by calling the routine `WaitNextEvent()`. If an event is forthcoming from `WaitNextEvent()`, the event loop next calls the appropriate function to handle the event. For example, if your application receives a keystroke (actually a key down event to the Mac), the action is passed to a function that might drop the character into a document window. Note that if certain windows are active (such as a Desk Accessory) or certain key combinations are pressed, different sets of handler code might be called to process the event. Continuing with our key down example, if you hold down the Command key while typing a character, the application instead calls functions that ultimately have a Menu Manager routine field the event. (Recall that a Command-key combination can be the keyboard equivalent for a menu choice.) A Mac application, in some instances, can be programmed to ignore certain events.

The basic structure of a Mac application is shown in Figure 3.33. A Mac application goes through its initialization phase and then runs in an event loop. As events trickle in, the event loop code checks to see what type of event occurred, and calls the corresponding function to handle the event. It

keeps doing this until the user signals the application to quit. At this point the application exits the event loop and performs any required clean up operations, such as saving files or discarding memory buffers.

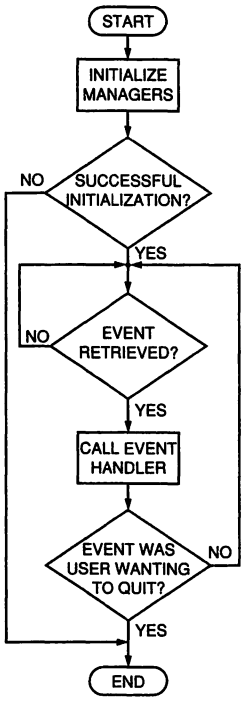


Figure 3.33 Structure of a Mac application.

An important distinction to make here is that events might occur in any order, and your program must be structured to deal with such disordered input. It shouldn't force the user through a gauntlet of dialog boxes that prompt for information. Also, because users aren't likely to explore every menu choice or dialog box setting, applications should provide reasonable defaults that help them get started. As an example of this, a word processor should default to a specific font (such as Times) and point size (say 12, for example) when displaying text. Along these same lines, any setting that the user might change frequently (such as the baud rate in a terminal emulator application) should be easy to find and change. If you're not familiar with this sort of user interface design, be sure to check out Apple's Human Interface Guidelines.

Code at Last

With the interface in place and a firm understanding of events, we can rewrite the munger program. Fire up CodeWarrior's IDE and create a new project. For a project name, type **munger.μ**. (Remember that the project name either must correspond to the resource filename "munger.π.rsrc" that we made with ResEdit, or in this case, we must manually add the resource file to the project.) Now pick Add File... from CodeWarrior's Project menu, and along the path CodeWarrior:Example Code PPC:MacMunger, open the file "Macmunger.c." Inside the Project window, double-click on the "Macmunger.c" file to open it with the built-in editor. In the Editor window, examine the following code:

```
#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Controls.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Devices.h>
#include <Memory.h>
#include <Events.h>
#include <Desk.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <ToolUtils.h>
#include <TextUtils.h>
#include <StandardFile.h>
#include <Errors.h>
#include <Resources.h>
#include <DiskInit.h>

/* Resource ID numbers */
#define LAST_MENU       3    /* Number of menus */
#define APPLE_MENU      128  /* Menu ID for Apple menu */
#define FILE_MENU       129  /* Menu ID for File menu */
#define EDIT_MENU       130  /* Menu ID for Edit menu */
#define RESOURCE_ID     127  /* Starting index into the
                                menu array */
```

```
#define ABOUT_BOX       1    /* About box menu item # in
                                Apple menu */

#define OPEN_FILE       1    /* Open item # in File menu */
/*---------------------*/  /* Separator line is item # 2 */
#define I_QUIT          3    /* Quit item # in File menu */

#define ABOUT_BOX_ID    128  /* Resource IDs for our windows
                                & dialogs */
#define STATUS_BOX_ID   129
#define ERROR_BOX_ID    130

/* Various constants */
#define NIL             0L
#define FALSE           false
#define TRUE            true

#define INIT_X          112  /* Coords for disk init dialog box */
#define INIT_Y          80

#define APPEND_MENU     0
#define CHAR_CODE_MASK  255
#define IN_FRONT        -1
#define NO_CURSOR       0L
#define ONE_FILE_TYPE   1
#define LONG_NAP        60L

#define CR              0x0D
#define LF              0x0A
```

As you can see, we intend to use more Managers this time, and consequently have more header files to include.

Next, we define the resource ID numbers of our menus and dialog boxes. These values come straight from our work in ResEdit. Look carefully at the menu item numbers in this section. These are values that the Menu Manager returns to the program when the user makes a menu choice. Notice that the menu item numbers start at 1, and that each separator line also counts as a menu item. If you add or remove items from a menu resource, the item numbers returned by the Menu Manager will change. You'll have to edit the

definitions here to match the new menu resource. To help keep this arrangement straight, notice how the `#defines` for the File menu are written so that they resemble the File menu layout. The rest of the section defines constants that we'll use elsewhere in the program, including Return and Line Feed.

Here are some function prototypes:

```
/* Function prototypes */
Boolean Do_Command (long mResult);
Boolean Init_Mac(void);
void Main_Event_Loop(void);
void Report_Error(OSErr errorCode);

/* Application-specific functions */
void Ask_File(void);
void Munge_File(short input, short output, unsigned char *fileName);

/* Globals */
MenuHandle  gmyMenus[LAST_MENU+1];   /* Handle to our menus */
EventRecord gmyEvent;           /* Holds event returned by OS */
WindowPtr   geventWindow;       /* Our private window */
Boolean     guserDone;          /* Indicates user wants to quit */
CursHandle  gtheCursor;         /* Current pointer icon */
short       gwindowCode;
WindowPtr   gwhichWindow;       /* The window that got an event */

OSType      gfileCreator = {'MUNG'};  /* Output file's creator */
OSType      gfileType = {'TEXT'};     /* Output file's type */
```

You'll recognize some basic functions here, such as `Init_Mac()`, `Do_Command()`, and `Main_Event_Loop()`, whose purpose is obvious. Also, we have a function that asks for a file, and—of course—a function to munge the file's contents. We also declare some globals here. The global `gmyMenus[]` is an array of handles that will point to menu records. Menu records are data structures that the Menu Manager builds to manage menus, somewhat like the data structures the Window Manager uses for windows. The `gmyEvent` global contains an event record, which is a data structure that describes the type of event passed to the application. The globals `gfileType` and `gfileCreator` contain the type and creator information for munger's output file.

Background Info

Everyone has his or her own style for writing code. I'll explain my style here, not because it's superior, but so that you'll quickly understand what the code is doing. To prevent confusion between the Mac Toolbox routine names and the program's function names, I use underscores in the program function names. So, `StandardGetFile()` is a Toolbox routine, while `Ask_File()` is a function that I wrote. Variable names begin with a lowercase letter, such as `fileName`, unless it's a global variable. Global variable names begin with a lowercase g, such as `gmyEvent`. Program constants are all uppercase, such as `LAST_MENU`, unless it's a well-publicized constant defined by Apple, such as `everyEvent` or `watchCursor`. Lately, Apple has been preceding their constants with a lowercase k, such as `kCoreEventClass`, which helps identify them. Feel free to use a style that works for you. Just be consistent, and always comment your code.

The First Function

Now it's time to look closely at the first function in "Macmunger.c":

```
void Report_Error(OSErr errorCode)
{
unsigned char errNumString[8];

   NumToString((long) errorCode, errNumString);
   ParamText(errNumString, NIL, NIL, NIL);
   StopAlert(ERROR_BOX_ID, NIL);
} /* end Report_Error() */
```

This is our minimalist error reporting function. When a Toolbox routine returns an error code, we pass it to `Report_Error()`. Inside `Report_Error()`, we use the Toolbox routine `NumToString()` to convert the error code to a displayable text string. The resulting Pascal string is then passed to `ParamText()`, whose job is to insert up to four strings inside a window. Because we have only one string to display, `ParamText()`'s other three arguments are `NIL`. How does `ParamText()` know where to place each text string? Recall that when we made the alert resource's DITL item for munger, we typed in "I/O error, ID = ^0." The ^0 is the placeholder for this string. `ParamText()` substitutes the placeholder text with the string in `errNumString`, staying within the rectangle defined

by DITL item. After `ParamText()` does the insertion, we call `StopAlert()` to create the stop alert window. An example of how the alert box appears is shown in Figure 3.34.

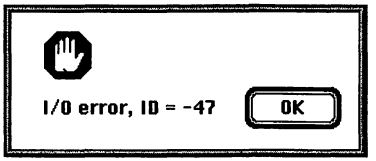


Figure 3.34 The `Report_Error()` alert box.

As error functions go, this is adequate for our work. If you get the stop alert, you can open the "errors.h" file from within the Metrowerks CodeWarrior editor and search for the error code to get an idea as to what went wrong. If you plan to unleash this program upon unsuspecting users, be nice to them and write an error reporting function that provides an explanation of the problem and suggests remedies. Don't dump a cryptic error ID number on the screen when trouble strikes.

Munger Code, Revisited

Let's examine the file munging code next:

```
void Munge_File(short input, short output, unsigned char *fileName)
{
long           amount;
unsigned char  buffer;
short          crflag;
long           icount, ocount;
unsigned char  inNumString[12], outNumString[12];
DialogPtr      statusDialog;

   amount = 1L;
   crflag = 0;
   icount = 0;
   ocount = 0;
   while (FSRead(input, &amount, &buffer) == noErr)
      {
      icount++;                    /* Bump input char counter */
      switch (buffer)              /* What char was read? */
```

```
        {
        case CR:
           if (crflag >= 1)  /* Two in a row, end of paragraph */
              {
              FSWrite(output, &amount, &buffer);  /* Write two
                                                     CRs */
              FSWrite(output, &amount, &buffer);
              crflag = 0;          /* Reset the flag */
              ocount++;
              } /* end if */
           else
              crflag++;     /* Bump the flag, and toss the CR */
        break;  /* end case CR */
        case LF:            /* Toss LF, but don't touch crflag */
        break;  /* end case LF */
        default:
           FSWrite(output, &amount, &buffer);
           ocount++;
           crflag = 0;      /* Clear the flag */
        } /* end switch */
     } /* end while */

/* Display processing statistics */
   if ((statusDialog = GetNewDialog(STATUS_BOX_ID, NIL,
                                    (WindowPtr) IN_FRONT)) != NIL)
      {
      NumToString(icount, inNumString);    /* Convert bytes read
                                              to string */
      NumToString(ocount, outNumString);
      ParamText (fileName, inNumString, outNumString, NIL);
      DrawDialog(statusDialog);
      Delay (120L, NIL);
      DisposDialog(statusDialog);
      } /* end if != NIL */
   else
      SysBeep(30);


} /* end Munge_file() */
```

`Munge_File()` accepts several arguments: an input file reference number, an output file reference number, and a pointer to a string containing the input filename. Since computers prefer to handle things as numbers, the File

Manager provides reference numbers for files that you open for reading or writing. These reference numbers remain valid as long as the files are open and you pass them to File Manager routines that perform the actual I/O. As you probably suspect, the function `Ask_File()` obtains these file reference numbers and then calls `Munge_File()`.

We use a `while` loop to read bytes with `FSRead()`, and then write bytes using `FSWrite()`, two other File Manager routines. If you compare this loop to the original munger.c code in Chapter 2, you'll see that the two are very similar, with `FSRead()` replacing `getc()` and `FSWrite()` replacing `fputc()`.

After the loop completes, we briefly display the processing statistics in a dialog box. To display these numbers, we fetch the `STATUS_BOX_ID` dialog resource using the routine `GetNewDialog()`. Like creating a window, we check to see if `GetNewDialog()` was successful at this. If it was, we convert the values in `icount` and `ocount` to strings. We pass these strings, plus the input filename, to `ParamText()` for inclusion in the dialog box. These strings will be substituted for the placeholders in the status box's DITL items, the same way it occurs in `Report_Error()`'s alert box. With the dialog box's contents set up, we call `DrawDialog()` to display the window. Next, we use `Delay()` to wait for two seconds. (`Delay()` waits for intervals of time called ticks, which are sixtieths of a second; 120 *ticks* is therefore two seconds.) Finally, we remove the dialog box and we're done.

Background Info

Seasoned Mac programmers will notice that we test for a failure by examining the pointer returned by `GetNewDialog()` to see if it is `NIL`. This type of check didn't work with earlier versions of the Mac OS. That's because these versions of `GetNewDialog()` would return a trash value if it failed. The workaround was to use a Resource Manager routine to see if the dialog resource existed before calling `GetNewDialog()`, like so:

```
if (GetResource('DLOG', ABOUT_BOX_ID) != NIL)
   {
   theDialog = GetNewDialog(ABOUT_BOX_ID, NIL,
                            (WindowPtr) IN_FRONT);
   ModalDialog(NIL, &itemHit);
   DisposDialog(theDialog);
```

continues

continued

```
    } /* end if != NIL */
else
    SysBeep(30);
```

Thanks to improvements to the Dialog Manager in System 7, we can use one consistent algorithm to test the results of Window and Dialog Manager routines.

There are a couple of things to note here. First, we don't do much error checking on the file I/O. This is so that you can examine the code easily, and verify that we're still using our basic algorithm here. Don't worry about this; we'll add this error-checking when we add high-level events to munger later. The other thing is that this code isn't very efficient, reading and writing only one byte at a time. `FSWrite()` automatically buffers some of the data during output, which improves performance somewhat. However, for faster I/O you would set `amount` to a large value so that `FSRead()` would read lots of data into a big buffer, process that buffer's contents, and then have `FSWrite()` write out large sections of the buffer. However, for my needs, the performance was adequate so that it wasn't worth the extra effort to improve the application's speed.

Input and Output Filenames

The next function to write is one that queries the user for input and output filenames, opens them, and supplies `Munge_File()` with the file reference numbers. This function is `Ask_File()`, whose code follows.

```
void Ask_File(void)
{
unsigned char     fileName[14] = {"\pMunge.out"};
short             inFileRefNum, outFileRefNum;
OSErr             fileError;
short             oldVol;
SFTypeList        textType = {'TEXT'};

StandardFileReply inputReply, outputReply;

/* Open the input file */
   StandardGetFile(NIL, ONE_FILE_TYPE, textType, &inputReply);
```

```
    if (inputReply.sfGood)
       {
       GetVol (NIL, &oldVol);           /* Save current volume */
       if ((fileError = FSpOpenDF (&inputReply.sfFile, fsCurPerm,
                                   &inFileRefNum)) != noErr)
          {
          Report_Error(fileError);
          return;
          } /* end if error */

/* Open the output file */
       StandardPutFile ("\pSave text in:", fileName, &outputReply);
       if (outputReply.sfGood)
          {
          SetVol(NIL, outputReply.sfFile.vRefNum);
          fileError = FSpCreate(&outputReply.sfFile, gfileCreator,
                                gfileType, smSystemScript);
          switch(fileError)  /* Process result from File Manager */
             {
             case noErr:
             break;
             case dupFNErr:  /* File already exists, wipe it out */
                if ((fileError = FSpDelete(&outputReply.sfFile))
                      == noErr)
                   {
                   if ((fileError = FSpCreate(&outputReply.sfFile,
                                              gfileCreator,
                                              gfileType,
                                              smSystemScript))
                                              != noErr)
                      {
                      Report_Error(fileError);
                      FSClose (inFileRefNum);
                      SetVol(NIL, oldVol);
                      return;
                      } /* end != noErr */
                   } /* end if == noErr */
                else
                   {
                   Report_Error(fileError);
                   FSClose (inFileRefNum);
                   SetVol(NIL, oldVol);
                   return;
```

```
                } /* end else */
            break;   /* end case dupFNErr */
            default:     /* Unknown error, try to abort cleanly */
               Report_Error(fileError);
               FSClose (inFileRefNum); /* Close the input file */
               SetVol(NIL, oldVol);    /* Restore original vol. */
               return;
            } /* end switch */

/* Open data fork */
         if (!(FSpOpenDF (&outputReply.sfFile, fsCurPerm,
                          &outFileRefNum)))
            {
            gtheCursor = GetCursor(watchCursor);   /* Change the
                                                      cursor */
            SetCursor(&**gtheCursor);
            Munge_File (inFileRefNum, outFileRefNum,
                        (unsigned char *) inputReply.sfFile.name);
            FSClose (outFileRefNum);
            SetCursor(&qd.arrow);         /* Restore the cursor */
            } /* end if !fileError */
         FlushVol (NIL, outputReply.sfFile.vRefNum);
         } /* end if outputReply.sfGood */
      FSClose (inFileRefNum);
      SetVol(NIL, oldVol);
      } /* end if inputReply.sfGood */


} /* end Ask_File() */
```

This code looks pretty scary, but it's not. We do a lot of error checking in this function because this is where the goofs that wipe out entire files can happen. First, `Ask_File()` uses the Toolbox routine `StandardGetFile()` to query the user for an input filename. The arguments `ONE_FILE_TYPE` and `textType` presented to `StandardGetFile()` have this routine filter out all file types but one, the 'TEXT' type. This averts potential fireworks by eliminating the possibility of accidentally opening a file loaded with binary data. When the routine returns, the file information is packaged in a `StandardFileReply` data structure. A part of this structure, the Boolean `sfGood`, indicates whether the contents of `StandardFileReply` are valid—that is, whether the user actually picked a file. We stop processing if `sfGood` is `FALSE`, because this occurs only when the user clicks on Cancel, which means they decided against munging a file.

If `sfGood` is `TRUE`, the program proceeds to open the file. First, we save the current default volume number using `GetVol()`. We do this because we will make the volume where the output is directed the current default volume temporarily. This way the actions of all File Manager routines apply to this specific volume, which might be another hard drive on the system or a shared Mac on the network. Then we use `FSpOpenDF()` to open the file's data fork. If there are no problems, `FSpOpenDF()` supplies a file reference number to be used with subsequent File Manager calls. Next, the program prompts the user for an output filename, using `StandardPutFile()`. The variable `fileName` provides a default name of "Munge.out" that `StandardPutFile()` offers when it displays the Standard File dialog. Again, we check `sfGood` to ensure that the user typed a filename (or used the default name) and clicked OK. If that's the case, then we set the output file's volume as the default volume.

While the tests for opening a file for input are simple, opening a file for output is anything but. For example, it's possible that the filename the user typed matches the name of a file that already exists in the folder. Fortunately, `StandardPutFile()` does this check for us and even tosses a dialog box on the screen, as shown in Figure 3.35, that warns of the conflict. However, if the user clicks on Replace, it's up to us to delete the existing file. We use a `switch` statement to deal with this situation and other errors.

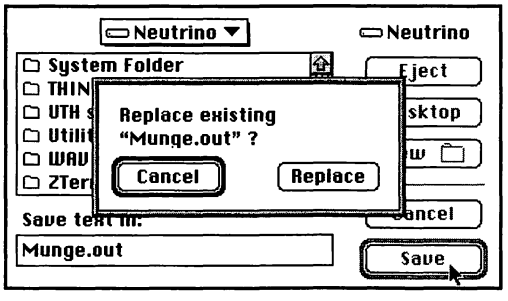


Figure 3.35 The name conflict dialog box.

First we attempt to create the file using `FSpCreate()`. If a duplicate filename error (-48) occurs, we delete the file with `FSpDelete()`, then try `FSpCreate()` again. If there are problems with these actions, or the first call to `FSpCreate()` happens to return an unexpected error code, we simply stop the operation. This is accomplished by calling `Report_Error()`, closing the input file (which we had opened), restoring the default volume number, and exiting `Ask_File()`.

This might seem like a drastic response, but when there's a disk full of files at risk, it's better to play it safe.

Assuming everything has worked flawlessly so far, `FSpCreate()` makes an output file of the requested creator and type, and the data fork is opened using `FSpOpenDF()`. If there are no errors, `FSpOpenDF()` returns a file reference number for the output file. Now we have all the information required by `Munge_File()`, since we can obtain the input filename string from `sfFile.name`, which is part of `StandardFileReply`.

Before we call `Munge_File()`, we fetch the stopwatch cursor icon using `GetCursor()`. The program places it on-screen by calling `SetCursor()`, to indicate that it is busy processing a file. Then, the program calls `Munge_File()`. When the function returns, the program sets the cursor back to an arrow. Now all that's left is the clean up. The output and input files are closed, and `FlushVol()` is called to update the volume information for the new file. Finally, the original default volume number is restored.

Background Info

Old timers will recognize that System 7's standard `GetFile()` and `StandardPutFile()` are similar to the old standard File Manager calls, `SFGetFile()` and `SFPutFile()`. The differences between the two sets of routines are minor, except for the type of reference number returned. These old routines are still supported for compatibility.

Basic Application Functions

Now it's time to look at some of the basic application functions that implement the user interface. Let's start with the code that handles menu commands:

```
Boolean Do_Command (long mResult)
{
unsigned char  accName[255];
short          itemHit;
Boolean        quitApp;
short          refNum;
DialogPtr      theDialog;
```

```
short          theItem, theMenu;
GrafPtr        savePort;      /* place to stow current GrafPort
                                 when Desk Accessory (DA) is
                                 activated */

   quitApp = FALSE;           /* Assume Quit not chosen */
   theMenu = HiWord(mResult); /* Extract the menu selected */
   theItem = LoWord(mResult); /* Get the item on the menu  */

   switch (theMenu)
      {
      case APPLE_MENU:
         if (theItem == ABOUT_BOX)          /* Describe ourself */
            {
            if ((theDialog = GetNewDialog(ABOUT_BOX_ID, NIL,
               (WindowPtr) IN_FRONT)) != NIL)
               {
               ModalDialog(NIL, &itemHit);
               DisposDialog(theDialog);
               } /* end if != NIL */
            else
               SysBeep(30);
            } /* end if theItem == ABOUT_BOX */
         else                       /* It's a DA */
            {
            GetPort(&savePort);   /* Save port (if DA doesn't) */
            GetMenuItemText(gmyMenus[(APPLE_MENU -
                                 MENU_RESOURCE)],
                                 theItem, accName);
            refNum = OpenDeskAcc(accName);   /* Start it */
            SetPort(savePort);    /* Done, restore the port */
            }
      break; /* end APPLE_MENU case */

      case FILE_MENU:
         switch(theItem)
            {
            case OPEN_FILE:
               Ask_File();        /* Obtain file info & process */
               break;
            case I_QUIT:          /* User wants to stop */
               quitApp = TRUE;
            break;
```

```
            } /* end switch */
        break; /* end FILE_MENU case */

        case EDIT_MENU:              /* Pass events to OS */
           SystemEdit(theItem - 1);
        break;
        default:
          break;
        } /* end switch */

    HiliteMenu(0);                   /* Switch off highlighting on
                                        the menu just used */
    return quitApp;
} /* end Do_Command() */
```

The `Do_Command()` basically takes a menu choice passed to it by the main event loop, and uses `switch` statements to route program execution to the appropriate handler code. This is accomplished by reducing the menu choice value in `mResult` into components using the `HiWord()` and `LoWord()` Toolbox routines. These components consist of the menu chosen, which is stored in `theMenu`, and the item on that menu, which is stored in `theItem`. For example, if someone using munger selected Quit from its File menu, `theMenu` would be 129 and `theItem` would be 3.

The first `switch` statement uses `theMenu`'s value to branch to a code section corresponding to that particular menu. Here a second `switch` statement uses `theItem`'s value to pick the function responsible for that specific menu item. Depending upon the number and structure of an application's menus, these switch statements can be sparse or complex.

For the Apple menu, if the About Box item is selected, then munger displays the dialog box we constructed in ResEdit. As usual, we check to see if `GetNewDialog()` encountered difficulties making the window. If not, `ModalDialog()` fields all events, keeping the About Box on the screen until the OK button is clicked or Return is pressed. If another item is picked on the Apple menu, its name is extracted using `GetMenuItemText()`. This name is passed to `OpenDeskAcc()`, which opens the Desk Accessory, application, document, or alias file in the Apple Menu Items folder. Note that we do some grafport housekeeping, just in case.

Background Info

In pre-System 7 versions of the Mac OS, the only items in the Apple menu were small utility programs called Desk Accessories that were embedded in the System file. They were actually a special type of driver so that they could run concurrently with the application in the original single-tasking environment. With the advent of cooperative multitasking under MultiFinder in System 6.0.x, the Mac OS treated Desk Accessories as applications, although the code and location of Desk Accessories didn't change. System 7 altered this arrangement further by creating an Apple Menu Items folder where the Desk Accessories appear as separate files. Not only that, but applications, documents, and the aliases to remote volumes can be placed in this folder and can be picked from the Apple menu. `OpenDeskAcc()`'s role, which was formerly limited to starting drivers in the System file, has thus expanded to deal with a variety of objects located in a special folder.

Munger's File menu is pretty simple. If the Open item was picked, we just call `Ask_File()`, and let it handle the job. If Quit was chosen, we set the variable `quitApp` to `TRUE`, to signal the main event loop that it's time to stop. The Edit menu is even simpler. As mentioned earlier, it's mostly a placeholder used to trickle certain events to other applications. The program calls `SystemEdit()`, which checks to see if the Edit menu selection (such as a Paste command) should be passed to a Desk Accessory or handled by the program itself. This is a holdover from the single-tasking days when only one application could run at a time, yet could support one or more Desk Accessories running symbiotically within it.

Just before `Do_Command()` exits, it performs some screen maintenance. When you make a menu selection, the Menu Manager highlights the menu's title. This serves as a visual cue that the application is doing something, especially if the chosen operation happens to be a lengthy one. (Ideally, the programmer also changes the cursor to a stopwatch, or some other busy indicator.) Once the operation completes, we call `HiliteMenu(0)` to restore the menu title's appearance. Finally, `Do_Command()` returns the value of `quitApp` to the main event loop.

Main Event Loops

Speaking of main event loops, it's time to check it out:

```
void Main_Event_Loop(void)
{
Point    where;

   FlushEvents(everyEvent, 0);   /* Clear out left over events */
   guserDone = FALSE;

   do
      {
      if (WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR))
         {                       /* We have an event... */
         switch(gmyEvent.what)   /* Field each type of event */
            {
            case mouseDown:      /* In what window, and where?? */
               gwindowCode = FindWindow(gmyEvent.where,
                                        &gwhichWindow);
                  switch(gwindowCode)
                  {
                  case inSysWindow: /* It's a Desk Accessory */
                     SystemClick(&gmyEvent, gwhichWindow);
                  break;
                  case inDrag:      /* Drag the window */
                  break;
                  case inGrow:      /* Change the window's size */
                  break;
                  case inContent:   /* Bring window to front if
                                       it's not */
                  break;
                  case inMenuBar:   /* In a menu, handle the
                                       command */
                     guserDone = Do_Command(MenuSelect(gmyEvent.where));
                  break;
                  } /* end switch gwindowCode */
               break; /* end mouseDown */
            case keyDown:
            case autoKey:   /* Command key hit, pass to MenuKey */
               if((gmyEvent.modifiers & cmdKey) != 0)
                  guserDone = Do_Command(MenuKey((char)
                                       (gmyEvent.message
```

```
                                    & CHAR_CODE_MASK)));
                break; /* end key events */
            case updateEvt:      /* Update the window */
                  gwhichWindow = (WindowPtr) gmyEvent.message;
                break;
            case diskEvt:        /* Handle disk insertion event */
               if (HiWord(gmyEvent.message) != noErr)
                  {
                  DILoad();
                  where.h = INIT_X;
                  where.v = INIT_Y;
                  DIBadMount(where, gmyEvent.message);
                  DIUnload();
                  } /* end if != noErr */
            break; /* end disk event */
            case activateEvt:       /* Activate event */
                  gwhichWindow = (WindowPtr) gmyEvent.message;
            break;
            default:
            break;
            } /* end switch gmyEvent.what */
         } /* end if on next event */
      } /* end do */

   while (guserDone == FALSE);      /* Loop until told to stop */
} /* end Main_Event_Loop() */
```

`Main_Event_Loop()` is the heart of the application. The program tours the loop in this function for the life of the application, retrieving events from the operating system queue and responding to them. The loop stops only when the user selects Quit from the File menu.

This function starts by clearing out any leftover events using `FlushEvents()`, and then sets `guserDone` to FALSE so that the `do` loop cycles permanently. Inside the `do` loop, the program calls `WaitNextEvent()` periodically, looking for events to handle. The first argument to this Toolbox routine is the event mask, which determines the types of events you want returned to the application. The munger program allows all of them. This mask can be modified in eclectic applications to filter out certain events. The second argument is a pointer to an event record, the data structure containing information on the type of event received, the pointer's screen location (necessary if the event was a mouse down), and any modifier information. *Modifiers* are the

Command, Option, Shift, and Control keys. When these keys are pressed, typically during a key down or mouse down event, they modify the meaning of the event, hence their name. You're already familiar with one modifier key: pressing Command and another key transforms a key down event into a menu selection.

The `LONG_NAP` constant informs `WaitNextEvent()` that munger should sleep for one second intervals, thereby yielding processor time to other applications. Like `Delay()`, this value is in tick intervals. This might seem like a lot of time to offer to the rest of the system, but munger isn't doing a time-critical background task such as a ZMODEM download, or copying a file across a network. Because munger does no background processing, the `if` statement around `WaitNextEvent()` locks out any null events. In this case, `LONG_NAP` simply serves as a placeholder in the routine. The constant `NO_CURSOR` tells the Mac OS that no special pointer handling is required.

Important

It's very important that your application periodically surrender the processor to other applications. (That is, `LONG_NAP` should never equal 0.) System 7 currently uses cooperative multitasking, where applications agree to share processing time amongst themselves. If your application fails to share time with other applications, background processing ceases because those applications can't get processor time to run.

The context switch to another application is handled through the `WaitNextEvent()` routine, so it must be called periodically to ensure that these switches occur. This isn't a problem when program execution is in the main event loop. However, functions called by the main event loop in response to a user command might keep program execution out of the loop long enough so that these application switches fail to happen regularly. For example, if `Munge_File()` performs disk I/O to a floppy—a slow peripheral device—other background applications get starved for processor time until the slow file I/O completes, and execution returns to munger's main event loop. The solution is to have the function periodically call `WaitNextEvent()` itself as it runs. Cooperative multitasking dictates that this type of program design must be used, since it's up to the application to relinquish control to other applications frequently.

Historically, applications used the original event dispatching routine `GetNextEvent()`, and it's still supported for compatibility. However, it's preferable to use `WaitNextEvent()`, since this routine is better suited for System 7's multitasking environment. For example, `WaitNextEvent()` provides the sleep argument, while `GetNextEvent()` doesn't.

Future Directions

Apple will release Copland, a microkernel-based OS, in the next year or so. One of its most important features is that task and process activity is handled by a preemptive time scheduler. That is, switches between applications occur at regular intervals based on the system clock, rather than how often `WaitNextEvent()` gets called in an application. This radical change in the OS design offers two benefits. First, a poorly written application can't hog processor time and stall the system. Second, it simplifies the programmer's job because she doesn't have to deal with OS issues when writing functions, as is the case with cooperative multitasking.

The event loop code has an arrangement similar to `Do_Command()`, where switch statements zero in on the function that deals with a specific event. The first switch statement uses the information in `gmyEvent.what` to jump to the code section for that event type.

Background Info

The categories of event types defined by the Mac OS are: mouse down, mouse up, key down, key up, auto key, update, disk insertion, activate, high-level, null events, and OS events. Most of these events are self-explanatory, but a brief description of the others is in order. The auto key event occurs when a key is held down long enough to begin repeating the character. The disk insertion event indicates a floppy or other removable media has been placed in a drive. The update event signals an application to redraw the contents of a specific window. Update events occur when other windows cover the application's window(s) temporarily, perhaps

continues

continued

because of an application context switch or because of a dialog box. The activate event informs the application that a certain window has been clicked on with the mouse, and if it isn't the current active window, it must be made so. Null events indicate that the user has done nothing; there are no other events to report. The application can either discard this type of event, or perform some background processing, such as blinking an I-beam cursor in a window with text. OS events are used for window maintenance and Clipboard data conversion when your application switches into the background or foreground. Since munger has no windows, and doesn't use the Clipboard, we ignore this event type.

Depending upon the event type, yet another `switch` statement might be used to further refine what code should respond to the event. For example, the mouse down event section uses a second `switch` statement to determine if the mouse was clicked on a window, the desktop, a Desk Accessory, or in the menu bar. Conversely, dealing with disk insertion events is a straightforward procedure, and so its section just has the code that handles the event.

Let's look closely at how events are dealt with. For a mouse down event, the code has `FindWindow()` evaluate where the mouse click occurred. We provide `FindWindow()` with this point from `gmyEvent.where`, which is part of the event record that contains the mouse position. `FindWindow()` returns a code that describes the part of the window clicked on, such as the title bar section of a window, the content region (where the application-specific information appears in the window), its size box (the box at a window's lower right, used to resize the window), or elsewhere. Elsewhere can be the on-screen desktop or the menu bar. If the code corresponds to a window element, `FindWindow()`'s second argument returns a pointer to that window. We use `FindWindow()`'s results in a `switch` statement to hop to the appropriate handler code. Since munger doesn't use a window, most of the handlers in this `switch` statement are stubs.

If the mouse click occurred in a system window (a Desk Accessory), we call `SystemClick()` to forward the event to it. This is one of those vestigial routines used for compatibility with older software. If the click happened in the menu bar, we hand the event off to `Do_Command()` for processing.

Key down and auto key events are treated the same way. Again, since munger doesn't use a document window, processing keystrokes is fairly simple. We peek at the modifier field (`gmyEvent.modifiers`) for each key event record. If the Command key wasn't pressed, then we toss the event in the bit bucket. If it was, the key event might be a menu's keyboard equivalent. First, we extract the character out of the `message` field of the event record. We use `CHAR_CODE_MASK` to do this, because this field is an amalgam of the key's character code, a virtual key code (a special code used to identify a physical key on the keyboard), and the address of the keyboard on the Apple Desktop Bus (ADB). We pass the character to `MenuKey()`, which maps it to the menu and menu item with the corresponding keyboard equivalent. `MenuKey()` returns a match in a format that we can simply pass along to `Do_Command()` to complete.

Like the mouse down window handlers, because the activate and update events pertain only to windows, we only put code stubs in the event loop for them. If your application uses windows, you'll have to flesh out this code.

The disk insertion event actually turns out to be a critical one for munger. Suppose someone decides to save the munged output to a blank floppy? When such a disk insertion event occurs, munger's handler code springs into action.

It checks the event's `message` field for an error code the Mac OS might return when it attempts to mount the volume (floppy). If a formatted floppy was inserted, the Mac OS mounts it so that a floppy disk icon appears on the desktop, and no error is reported. If there was a mount error, we retrieve the event and call `DILoad()` to load the Disk Initialization Manager. We pass the event's message field to `DIBadMount()`, a routine used to initialize (or format) volumes. Message supplies `DIBadMount()` with the error code and the drive number. `DIBadMount()` places a dialog box on the screen, asking the user to initialize or eject the floppy. The user presumably initializes the floppy and `DIBadMount()` exits. `DIUnload()` then removes the Disk Initialization Manager from memory, and the user has a fresh floppy on which to save munged files.

If munger didn't field this event, when the user poked a blank floppy into the drive, nothing would happen. The disk insertion event would remain queued until the user switched to another application (probably a database to look up my Internet address and rightfully complain). This application would handle the event, and the disk initialization dialog would appear unexpectedly,

further confusing the unhappy user. Although munger uses `DIBadMount()` to initialize a floppy disk, this routine can also initialize hard disks. See Inside *Macintosh: Files* for more information.

As the event loop completes its course, the variable `guserDone` is checked. If `Do_Command()` returns `TRUE` in response to a Quit command, this changes the state of `guserDone`. In this case the event loop quits, as does the application.

The Initialization Function

Now all that's left to do is examine munger's initialization function:

```
Boolean Init_Mac(void)
{
short i;

/* Lunge after all the memory we can get */
   MaxApplZone();

/* Make sure we've got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(NIL);

/* Loop to setup menus */
   for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
      {
      gmyMenus[(i - RESOURCE_ID)] = GetMenu(i);
                                    /* Get menu resource */
      if (gmyMenus[(i - RESOURCE_ID)] == NIL)
                                    /* Didn't get resource? */
         return FALSE;              /* No, bail out */
```

```
      }; /* end for */

/* Build Apple menu */
   AppendResMenu(gmyMenus[(APPLE_MENU - RESOURCE_ID)], 'DRVR');

/* Add the menus */
   for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
      InsertMenu(gmyMenus[(i - RESOURCE_ID)], APPEND_MENU);

   DrawMenuBar();
   InitCursor();                    /* Tell user app is ready */
   return TRUE;
} /* end Init_Mac() */
```

If you think this function looks similar to the "Hello world" example at the start of this chapter, you are correct. Notice that we've added code to initialize the Menu Manager, Dialog Manager, and TextEdit, a Manager that deals with simple text entry and editing. TextEdit is required to handle characters typed into the Standard File dialog box when `StandardPutFile()` asks for a filename.

We also have to set up the menus. First, we use a `for` loop to load the menu resources, using `GetMenu()`. This routine returns a handle to menu record, which we immediately stow in our `gmyMenus` array. We do some math to convert the menu resource ID into an array index. Because the initialization code runs only once, we can afford to do some extra calculations here. We also perform a fail-safe check to see that `GetMenu()` successfully locates the menu resources and returns valid handles to menu records. If there is a problem, `GetMenu()` returns `NIL`. In this case we simply abort the initialization process and have the function return `FALSE`.

Next, we use `AppendResMenu()` to construct the Apple menu. The `AppendResMenu()` routine searches any resource files open to the application for the requested resource type. It then adds the names of these resources to the specified menu. We specify the DRVR resource to collect the Apple menu items. Like the operation of `OpenDeskAcc()` routine, this resource type selection is a remnant of the pre-System 7 days when Desk Accessories were driver resources in the System file. However, `AppendResMenu()` now fetches the names of all the files in the Apple Menu Items folder, as well as the Desk Accessories.

With the Apple menu built, we use another for loop to add munger's own menus using `InsertMenu()`. Finally, we display the new menus using `DrawMenuBar()`, followed by `InitCursor()`, which sets the mouse pointer to an arrow to show the munger is ready.

The last thing left to do is type `main()`, and here it is:

```
void main(void)
{
   if (Init_Mac())
      Main_Event_Loop();
   else
      SysBeep(30);
} /* end main */
```

When the application launches, it calls the initialization function. If the function reports no problem (by returning `TRUE`), then execution proceeds to the main event loop. This is where munger runs until the user asks it to quit.

Build Munger

Now it's time to build munger. Add "Macmunger.c," and "munger.π.rsrc" to the project. Remove the placeholder items <replace me Mac>.c and <replace me>.rsrc, the library file MathLib, and all of the ANSI libraries from the project window. The project window should look like that in Figure 3.36.

Go to the PPC Project preferences panel and type `munger` for the application filename. Build the application and an application file, sporting the generic application icon and the name munger, should appear in the folder. Launch munger, and try it out.

Important

If munger beeps and quits immediately, there's a problem with its menu resources. First, check to see that the .rsrc filename matches the project filename. (That is, the project munger.μ should have a resource file named "munger.μ.rsrc.") Or, if the resource filename differs from the project name as in our example, confirm that the resource filename appears in the project window. If not, use Add Files... from the Project menu to put it there. If you're sure the resource file is there, the problem might be with

the resources themselves. Open the .rsrc file with ResEdit and make sure the ID numbers of the MENU resources match those defined in munger's source code. If not, correct the problem by changing the ID numbers in ResEdit, or editing "Macmunger.c." Note that Macmunger.c's initialization code relies on the MENU ID numbers to be in ascending order.

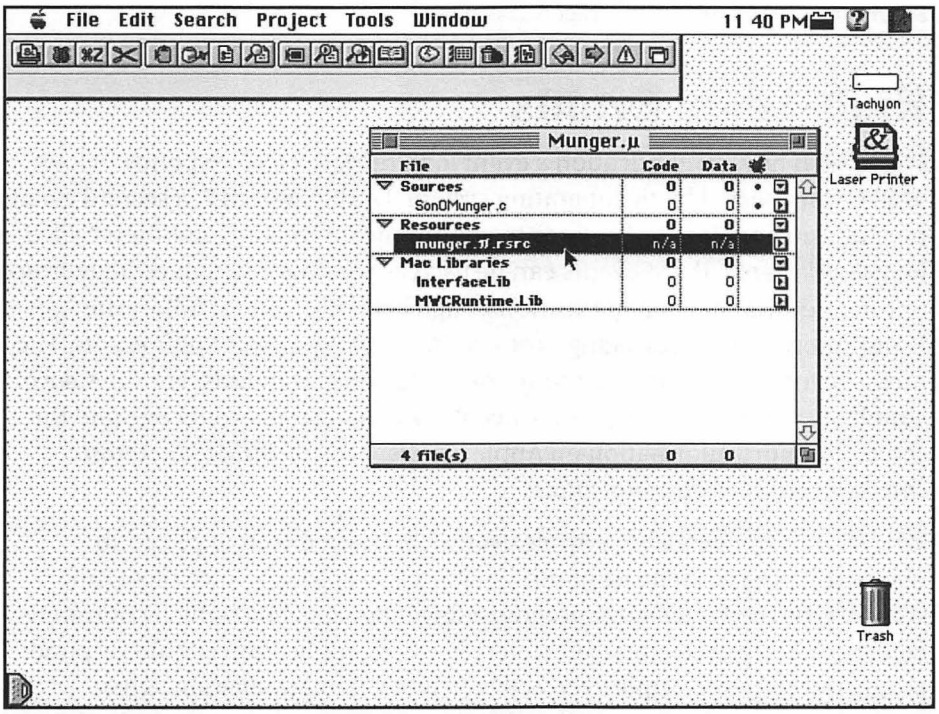


Figure 3.36 The Munger project window. Notice the resource file uses a different name.

Choose Open... from the File menu, and search for a file to munge. You'll see that the only files that appear in the Standard File dialog are folders or text files. Select a text file, such as the sample file in CodeWarrior:Code Examples PPC:Munger:PowerPC.txt and let munger have at it. When munger is done, you'll get the status dialog box (see Figure 3.37) that reports on the results of the filtering operation.

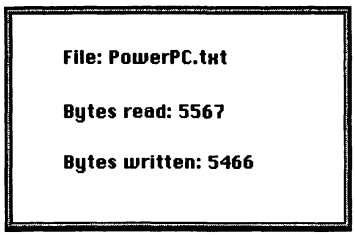


Figure 3.37 Munger's status report dialog box.

High-Level Events

We've seen how an application's event loop retrieves and responds to low-level events posted by the operating system. Under System 7, a second event mechanism enables applications to communicate with one another. Called high-level events, these events can be used as messages to request data from or provide data to other applications. High-level events that follow the Apple Event Interprocess Messaging Protocol (AEIMP) are called *Apple events.* The message format is defined by suites of published commands. For the sake of simplicity, we will consider Apple events and high-level events one and the same. For more information on Apple events, consult *Inside Macintosh: Interapplication Communication.*

Why should you care about Apple events? Because if your application responds to them, it can be controlled by a Power Mac's voice recognition software, or the AppleScript programming language, both which communicate through Apple events. At the very least, an application should respond to the four required Apple events, which are: Open Application, Open Documents, Print Documents, and Quit Application. Although this quartet of required events seem rather limited, a creative script can do a lot with them.

For example, you can write a program in AppleScript that searches a folder for the email you just downloaded, launches a word processor application (Open Application event), instructs it to print your email files (Print Documents event), and then stops the word processor (Quit Application event). The Finder, where possible, uses the required events to open documents and handle print requests. Of course, the applications have to be "savvy" (or understand) Apple events for the Finder to do this. A special resource in the application tells the Finder whether it's Apple event savvy or not. If not, the Finder uses older, pre-System 7 methods to start the application and handle the request.

One compelling reason to add high-level event support to munger is that it allows us to use System 7's neat drag-and-drop mechanism. That is, someone selects a text file icon with the mouse, drags it across the desktop, and drops it onto the munger icon. Munger launches, and through high-level event communications, opens the desired file and processes it. Let's add this capability to munger, since we only need one of the four required Apple events to implement it. While we're at it, we'll give munger a distinctive icon, and beef up the error checking in the `Munge_File()` function, as promised earlier.

Make Munger Handle High-Level Events

There are four key sections in munger that we have to change so that it handles high-level events. First, we've got to make our event loop code aware of this new type of event. Second, we need a mechanism that delivers these high-level events to the appropriate handler functions. Third, we need the handler code itself. Last but not least, we have to make the operating system aware that our application can deal with high-level events.

Begin by making a copy of "Macmunger.c." Select the "Macmunger.c" file and pick Duplicate from the Finder's File menu, or type Command-D. Rename the file copy "SonOMunger.c" and add it to the munger.μ project, while removing the original "Macmunger.c" from the project. Open "SonOMunger.c" with the editor to add a few more header files to the program. Beneath the other header files, type:

```
#include <AppleTalk.h>
#include <AppleEvents.h>
#include <EPPC.h>
#include <PPCToolBox.h>
#include <Processes.h>

struct AEinstalls
   {
    AEEventClass theClass;
    AEEventID theEvent;
    AEEventHandlerProcPtr theProc;
   };
typedef struct AEinstalls AEinstalls;

#define LAST_HANDLER      3    /* Number of Apple Event handlers - 1 */
```

Most of these header files define Apple event data structures and routines. "AppleTalk.h" is required because high-level events can be sent across the network to other computers. To communicate to other applications, Apple events also use certain Process Manager routines and so that a header file appears. The structure `AEInstalls` organizes certain Apple event data structures and the addresses of handler functions for installation in a dispatch table. `LAST_HANDLER` indicates how many of these handlers must be installed in the dispatch table. There are a few more function prototypes to define, too:

```
/* High-level Apple Event functions */
Boolean Init_AE_Events(void);          /* Install the handlers */

/* Post high-level event to the dispatch table */
void Do_High_Level(EventRecord *AERecord);

/* The four required handlers */
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein,
                                  ➥AppleEvent *reply, long refIn);
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein,
                                  ➥AppleEvent *reply, long refIn);
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein,
                                  ➥AppleEvent *reply, long refIn);
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein,
                                  ➥AppleEvent *reply, long refIn);

/* Note change! */
OSErr Munge_File(short input, short output, unsigned char *fileName);
```

There's the usual initialization function to install the handlers, a function to route the high-level events to the handlers, and the four handlers themselves. As part of its improved I/O checks, `Munge_File()` returns an error value now.

Modifying the Event Loop Code

Now let's start with the first item on the list, which is modifying the event loop code. Go to `Main_Event_Loop()`, and in the first switch statement (the one that deals with the event type), add:

```
case activateEvt:          /* Activate event */
   gwhichWindow = (WindowPtr) gmyEvent.message;
break;
case kHighLevelEvent:      /* Handle Apple Event */
   Do_High_Level(&gmyEvent);
```

```
        break;
        default:
        break;
        } /* end switch gmyEvent.what */
```

I've included a few of the surrounding source code statements so that you can recognize where to place the code. From this code you can see that high-level events are just another event passed to the application via the Event Manager. The operation of this new code is simple: When `WaitNextEvent()` retrieves a high-level event for us, we just call `Do_High_Level()` to handle it.

Delivering High-Level Events

Let's write `Do_High_Level()` next, since it's a portion of item two, the delivery mechanism. Type:

```
void Do_High_Level(EventRecord *AERecord)
{
   AEProcessAppleEvent(AERecord);
} /* end Do_High_Level() */
```

Was that tough, or what? The event record gets forwarded directly to `AEProcessAppleEvent()`. This routine uses information in the event record to determine what handler routine to call in the dispatch table. The application's dispatch table is searched first, followed by the system's dispatch table. A match is based on the event's class and event ID. If there is a match, `AEProcessAppleEvent()` calls the handler associated with that dispatch table entry. This brings up the question of what builds the dispatch table. For the answer, type:

```
Boolean Init_AE_Events(void)
{
OSErr err;
short i;
static AEinstalls HandlersToInstall[] =  /* The 4 required Apple Events */
   {
      {kCoreEventClass, kAEOpenApplication, (AEEventHandlerProcPtr)Core_AE_Open_Handler},
      {kCoreEventClass, kAEOpenDocuments, (AEEventHandlerProcPtr)Core_AE_OpenDoc_Handler},
      {kCoreEventClass, kAEQuitApplication, (AEEventHandlerProcPtr)Core_AE_Quit_Handler},
      {kCoreEventClass, kAEPrintDocuments, (AEEventHandlerProcPtr)Core_AE_Print_Handler}
   };

    for (i = 0; i < LAST_HANDLER; i++)
```

```
        {
        err = AEInstallEventHandler(HandlersToInstall[i].theClass,
                                    HandlersToInstall[i].theEvent,
              NewAEEventHandlerProc(HandlersToInstall[i].theProc),
                                                        0, FALSE);
        if (err)    /* If there was a problem, bail out */
           return FALSE;
        } /* end for */

    return TRUE;
} /* end Init_AE_Events() */
```

It's the responsibility of `Init_AE_Events()` to construct the table. The objects in the array `HandlersToInstall[]` correspond to the dispatch table elements of an event class, an event ID, and a pointer to a handler function. A simple `for` loop calls `AEInstallEventHandler()`, a routine that plugs these items into the table. If the routine reports an error, we pass a failure indicator (`FALSE`) back to `Init_Mac()` to halt the application.

Hazard

Don't overlook the `NewAEEventHandlerProc()` routine that's buried innocuously as an argument in the call to `AEInstallEventHandler()`! This routine is critical for the proper setup of the handler functions. Since the Power Mac's system software is a mixture of 680x0 and PowerPC code, it gets tricky for the operating system to know what type of code it will be running next when the thread of execution hops to another function. To combat this problem, Apple devised Universal Procedure Pointers, or UPPs. The UPPs describe to the Mixed Mode Manager what type of processor code (PowerPC or 680x0) the function uses, the number of arguments the function uses, and the programming language used to implement the function. The programming language distinction is necessary because C programs pass their arguments to a function in an order that's different from Pascal.

The C header files incorporate this UPP information for every Toolbox routine so that the programmer is normally unaware which routines are PowerPC code, and which are 680x0 code. For certain functions that you write, it's up to you to explain their nature to the Mixed Mode Manager by providing UPPs for them. Functions that fall in this category are external

functions (such as plug-in modules that might be a mixture of 680x0 or PowerPC code), or your own internal functions that get called by the operating system (such as our high-level event handlers). Such functions are termed *callback functions* because you first call a Toolbox routine, and in response to an action, the Mac OS calls a function back in your application. It's possible that a context (or mode) switch will occur because your function might be native code, whereas the OS routine performing the callback is 680x0 code. Or, conversely your 680x0 application's callback function gets called by a native Toolbox routine. If you fail to provide a UPP for these callback functions, the Mac OS can get terribly confused when it jumps to them. This is because the operating system doesn't know what processor code the function is written in, nor can it determine the size of the arguments used. If the Mac OS guesses wrong, the result is a spectacular crash.

The rule of thumb is: If a mode switch is involved, you need a UPP. Native PowerPC plug-in modules that add capabilities to a PowerPC application don't require UPPs because there is no mode switch involved.

If you're worried about getting bogged down in the details of writing a UPP, relax. The header files supply routines that do this work for you, especially when it's known that the operating system will be calling back into your application. `NewAEEventHandlerProc()` is such a routine; it constructs a UPP for those high-level event handlers whose addresses you supply. Don't forget to use this routine when setting up your handlers! For a more detailed explanation of what a UPP is, and how it works, see Chapter 4.

We call `Init_AE_Events()` as the SonOMunger initializes so that it is prepared to respond to Apple Events immediately once it is running. Go to the `Init_Mac()` function and type:

```
DrawMenuBar();

if (!Init_AE_Events())     /* Set up our high-level event
                              handlers */
   return FALSE;

InitCursor();              /* Tell user app is ready */
```

Again, I have included a few neighboring lines of code so that you get the idea of where to locate the function call. This completes item two.

Writing the Handlers

Item three on our list is writing the handlers. Enter the following code:

```
/* High-level open application event.  */
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein,
                                  AppleEvent *reply, long refIn)
{
    return noErr;
} /* end Core_AE_Open_Handler() */

/* High-level print event */
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein,
                                  AppleEvent *reply, long refIn)
{
    return errAEEventNotHandled;   /* No printing done here, so */
                                   /* no print handler. */
} /* end Core_AE_Print_Handler() */

/* High-level quit event  */
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein,
                                 AppleEvent *reply, long refIn)
{
    guserDone = TRUE;   /* Tell main event loop we want to stop */
    return noErr;
} /* Core_AE_Quit_Handler() */
```

The three handlers you see here are fairly simple. Notice that arguments passed to them are simply ignored. The Open Application Apple Event notifies the application to perform any start-up tasks required of it. For example, the application might create an untitled document window, or establish a connection to a database. Since SonOMunger's design of pipelining of data between two files is very focused, it doesn't need any start up tasks. Therefore, when munger receives an Open Application event, the function `Core_AE_Open_Handler()` reports a "no error" message back to the caller while doing nothing. Since SonOMunger doesn't do any printing, `Core_AE_Print_Handler()` responds with an error message that indicates SonOMunger can't field the Print Documents event. Upon the receipt of a Quit Application event, `Core_AE_Quit_Handler()` simply sets `guserDone` so that SonOMunger halts on the next tour of the event loop, and returns a "no error" message.

SonOMunger uses the Open Document Apple event to implement the drag-and-drop feature. When you drop a text file icon onto the SonOMunger icon, the Finder sends it an Open Document event that also contains the dropped filename. Drag-and-drop applications are generally expected to complete the job without further input from the user. That is, you drop a file on SonOMunger, and you expect a processed output file to appear. With that in mind, let's write the `Core_AE_OpenDoc_Handler()` function. Type:

```
/* High-level open document event */
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein, AppleEvent *reply,
➥long refIn)
{
short          i, j;
AEDesc         fileDesc;
OSErr          highLevelErr;
AEKeyword      ignoredKeyWord;
DescType       ignoredType;
Size           ignoredSize;
long           numberOFiles;
unsigned char  outFileName[64];
FSSpec         inFSS, outFSS;
short          inFileRefNum, outFileRefNum;
OSErr          fInErr, fOutErr, mungeResult;

   gtheCursor = GetCursor(watchCursor);  /* Indicate we're busy */
   SetCursor(&**gtheCursor);
   mungeResult = 0;               /* Clear so FOR loop operates */
/* Get parameter info (a list of filenames) out of Apple Event*/
   if (!(highLevelErr = AEGetParamDesc(messagein, keyDirectObject,
                                    ➥typeAEList, &fileDesc)))
      {
      if ((highLevelErr = AECountItems(&fileDesc, &numberOFiles))
           == noErr)
/* Count files */
         {
         for (i = 1; ((i <= numberOFiles) && (!highLevelErr) &&
                                          ➥(!mungeResult)); ++i)
            {
            if (!(highLevelErr = AEGetNthPtr(&fileDesc, i,
                                          typeFSS,
                                          &ignoredKeyWord,
                                           &ignoredType,
```

```
                                (char *)&inFSS,
                                sizeof(inFSS),
                                &ignoredSize))) /* Get name */
{
for (j = 1; (j <= inFSS.name[0]); j++)
                            /* Copy filename */
   {
   outFileName[j] = inFSS.name[j];
   } /* end for */
outFileName[j] = '.';      /* Tack '.out' on end */
outFileName[j + 1] = 'o';
outFileName[j + 2] = 'u';
outFileName[j + 3] = 't';
outFileName[0] = (j + 3);        /* Update string's length */
if (!(fInErr = FSpOpenDF(&inFSS, fsCurPerm,
      &inFileRefNum)))
   {
   if ((fOutErr = FSMakeFSSpec(DEFAULT_VOL, NIL,
                           outFileName, &outFSS))
                  == fnfErr)
      {
      if (!(fOutErr = FSpCreate(&outFSS,
                                gfileCreator,
                                gfileType,
                                smSystemScript)))
         {
         if (!(fOutErr = FSpOpenDF(&outFSS,
                                   fsCurPerm,
                                  &outFileRefNum)))
            {
            mungeResult = Munge_File(inFileRefNum,
                                     outFileRefNum,
                                        inFSS.name);
            FlushVol(NIL, outFileRefNum);
            FSClose(outFileRefNum);
            } /* end if !fOutErr */
         else
            Report_Err_Message("\pError opening
                                   output file");
         } /* end if !fOutErr */
      else
```

```
                    {
                    Report_Err_Message("\pError creating
                                        output file");
                    } /* end else */
                 } /* end if == fnfErr */
              else
                 {
                 if (fOutErr == noErr)  /* No error means a
                                           file already has
                                           that name */
                    Report_Err_Message("\pCan't write, file
                                       already exists");
                 } /* end else */
              FSClose(inFileRefNum);
              } /* end if !fInErr */
           else
              Report_Err_Message("\pError opening input
                                  file");
           } /* end if !highlevelErr */
        } /* end for */
     } /* end if == noErr */
  AEDisposeDesc(&fileDesc);     /* Dispose of the copy made */
                                /*   by AEGetParamDesc() */
  } /* end if !highLevelErr */

SetCursor(&qd.arrow);       /* Restore the cursor */
guserDone = TRUE;           /* We're done, stop the application */
return (highLevelEvent);
} /* end Core_AE_OpenDoc_Handler() */
```

The Open Document event definitely triggers some activity here. Starting at the top, `Core_AE_OpenDoc_Handler()` first slaps a stopwatch on the pointer to show that the application is busy. Next, the Apple event gets passed to `AEGetParamDesc()`, a routine whose arguments tell it to retrieve the data parameters from the Apple event record. These parameters are to be coerced (or massaged) into a data array termed a *descriptor list*, as specified by the `typeAEList` argument. This list is placed in a buffer created by `AEGetParamDesc()` and pointed to by `fileDesc`.

`NowAECountItems()` determines how many objects make up the descriptor list, which is the number of files dragged and dropped on SonOMunger. We use the value returned by this routine to set up a `for` loop that extracts each filename out of the descriptor list.

Two things to note here are: First, if an error occurs while extracting filenames from the descriptor list using `AEGetNthPtr()`, the loop terminates. Second, if there's an error during file processing, `mungeResult` goes non-zero, and the loop terminates. We do have to initially zero `mungeResult` so that the loop doesn't quit prematurely.

The `AEGetNthPtr()` routine actually obtains the filenames from the descriptor list. The routine's arguments instruct it to retrieve the descriptor list items as file system specification records (`typeFSS`), a format that's used by most System 7 File Manager routines. Any Apple event keyword and descriptor type information associated with the item is ignored (`ignoredKeyWord` and `ignoredType`). The largest data item returned from the descriptor list must be no larger than a file system specification record (`sizeof(inFSS)`), and the size of the data returned is ignored.

Once an input filename is obtained from the list, we tack an ".out" extension on it, creating our output filename. This eliminates the dilemma of what to name the output file without querying the user. Note that we should add a safety check here, to see that the filename is no larger than 27 characters. The reason is that Mac OS typically limits filenames to thirty-one characters in length. I should (but don't) perform a sanity check to ensure that the user hasn't passed a filename to SonOMunger that will be longer than this 31-character limit when we append the ".out" extension of the file.

To review, munger got the input filename from an Open Document Apple event that was the result of the user's drag and drop, and the output filename is derived from the input name. We use the `FSMakeFSSpec()` routine to make a file system specification record out of the derived output filename. The program then does the usual safety checks to ensure that the input and output files can be opened and written to properly, and gathers the file reference numbers. Finally, `Munge_File()` gets called.

If things proceed smoothly in `Munge_File()`, then the input and output files are closed, and the loop cycles to the next file. If for some reason `Munge_File()` encounters trouble, the error value it returns stops the loop so that the user can fix the problem. Note that we're trying to help the user do just that by improving the error reporting. The function `Report_Err_Message()` accepts a Pascal string that gets displayed in an alert window. Finally, we call `AEDisposeDesc()` to release the memory allocated by `AEGetParamDesc()` when it made a copy of the descriptor list for our use.

The code for `Report_Err_Message()` is:

```
void Report_Err_Message(unsigned char *errMess)
{

   ParamText(errMess, NIL, NIL, NIL);
   CautionAlert(ERROR_MESS_ID, NIL);
} /* end Report_Err_Message() */
```

This is a simple routine; it just takes a pointer to a Pascal string and passes this to `ParamText()`. `CautionAlert()` then places the message on-screen. The value here is in the descriptive messages that you can provide. This is because we know where the problem occurs in the handler code, so we've got a good idea as to what caused the error.

Last but not least, here's the improved `Munge_File()` function:

```
OSErr Munge_File(short input, short output, unsigned char *fileName)
{
long           amount;
unsigned char  buffer;
short          crflag;
long           icount, ocount;
OSErr          fInOutErr;
unsigned char  inNumString[12], outNumString[12];
DialogPtr      statusDialog;

   amount = 1L;
   crflag = 0;
   icount = 0;
   ocount = 0;
   while (FSRead(input, &amount, &buffer) == noErr)
      {
      icount++;               /* Bump input char counter */
      switch (buffer)         /* What char was read? */
         {
         case CR:
            if (crflag >= 1)  /* Two in a row, end of paragraph */
               {
               if (!(fInOutErr = FSWrite(output, &amount,
                                         &buffer)))
                  {
                  if ((fInOutErr = FSWrite(output, &amount,
                                    &buffer)) != noErr)
```

```
                    {
                    Report_Error(fInOutErr);
                    return fInOutErr;
                    } /* end if != */
                  } /* end if ! */
               else
                  {
                  Report_Error(fInOutErr);
                  return fInOutErr;
                  } /* end else */
               crflag = 0;          /* Reset the flag */
               ocount++;
               } /* end if */
            else
               crflag++;      /* Bump the flag, and toss the CR */
         break; /* end case CR */
         case LF:            /* Toss LF, but don't touch crflag */
         break;   /* end case LF */
         default:                  /* Write a character out */
            if ((fInOutErr = FSWrite(output, &amount, &buffer))
                                                   != noErr)
               {
               Report_Error(fInOutErr);
               return fInOutErr;
               } /* end if */
            ocount++;
            crflag = 0;            /* Clear the flag */
         break;
         } /* end switch */
      } /* end while */

/* Display processing statistics */
   if ((statusDialog = GetNewDialog(STATUS_BOX_ID, NIL,
                                  (WindowPtr) IN_FRONT)) != NIL)
      {
      NumToString(icount, inNumString);   /* Convert bytes read
                                             to string */
      NumToString(ocount, outNumString);
      ParamText (fileName, inNumString, outNumString, NIL);
      DrawDialog(statusDialog);
      Delay (120L, NIL);
      DisposDialog(statusDialog);
      } /* end if != NIL */
```

```
    else
        SysBeep(30);

    return fInOutErr;
} /* end Munge_file() */
```

This function is nearly identical to the original `Munge_File()`, except that the I/O routines are checked for errors. If a problem is detected, we simply call the original `Report_Error()` function, because it's hard to predict the types of problems that can occur at this level. We also pass back the error code to the caller so that action can be taken, as you saw in the Open Document handler code. This completes "SonOMunger.c." If you're confused about where the new functions went, examine "SonOMunger.c" on the CD-ROM. (The pathname is `CodeWarrior:Code Examples PPC:SonOMunger`.) Or, check the complete source listing in Appendix C.

However, we're still not finished. All that remains is to add some resources to SonOMunger that provide an alert box for the new error message function, and to inform the operating system that the new and improved SonOMunger is Apple Event-aware. Making SonOMunger appear high-level event savvy to the Mac OS will complete point number four, for those of you keeping score.

Making SonOMunger High-Level Event Savvy

In the CodeWarrior compiler, select Preferences from the Edit menu, and go to the PPC Project panel. Type **MUNG** for the Creator item. This assigns the application's Creator type, which must match the signature resource you'll make with ResEdit's bundle editor in a moment. Next, click on the checkmark icon next to the Size Flags item to activate the pop-up menu. Pick the isHighLevelEventAware item, and confirm that it is checked. Recall that earlier I mentioned that the Macintosh OS used a resource to determine if an application is high-level event savvy or not. The resource used for this determination is the SIZE resource, and we're setting the appropriate flag bit in it to indicate that SonOMunger can handle high-level events. If you fail to do this, the Mac OS assumes that SonOMunger can't handle high-level events and so none are ever sent to SonOMunger.

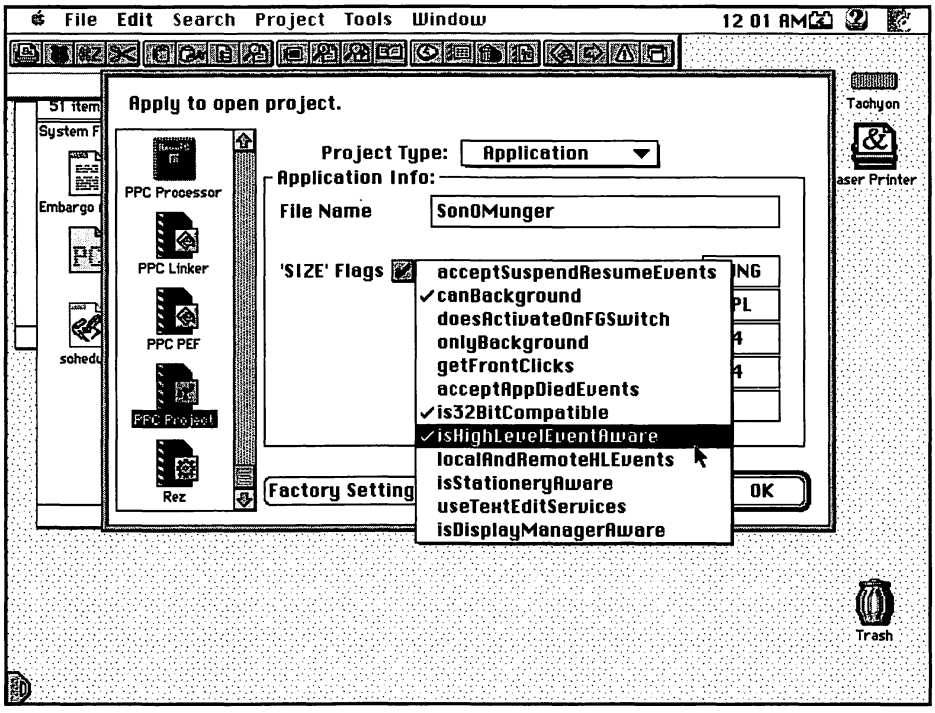


Figure 3.38 Setting a high-level savvy flag in the application's `SIZE` resource.

New Alerts

Now let's add the new resources SonOMunger requires. Start by double-clicking on the "munger.π.rsrc" file, which launches ResEdit. The window that displays the resource fork's contents appears. We'll make the alert box for the `Report_Err_Message()` function first.

Double-click on the ALRT resource icon. After the ALRT resource window opens, select Create New Resource from the Resource menu or type Command-K. When the alert editor window appears, change the DITL ID number to `131`. Next, pick Get Resource Info from the Resource menu or type Command-I. When the Info box opens, change the ID number to `131`. Click on this window's close box, and you have an alert resource with an ID number of 131, ready to edit.

Double-click on the window to bring up the dialog item (DITL) editor. Resize the window, add an OK button, and follow that with a static text box in the window's lower left. (Remember that the OK button needs to be DITL item 1, and that we need to allow space for the alert icon, which appears in the upper left window corner.) Simply type ^0 for the static text item in this box. You should have an alert window that resembles the one in Figure 3.39. Close all of the Editor windows, leaving only the one showing the view of the resource fork.

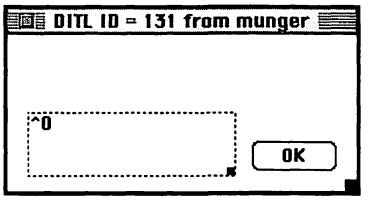


Figure 3.39 The `Report_Err_Message()` alert box.

Bundle Resource

To implement the drag-and-drop filtering, we must provide SonOMunger with a BNDL, or bundle resource. This resource gets its name because it describes the linkages among a so-called "bundle" of resources that are used to supply certain application characteristics to the Finder, and to display the application's icon on the desktop. Let's begin by building some of these bundled resources, beginning with an application icon for SonOMunger, and an icon for its output files.

In ResEdit, create a new resource. Select 'ICN#' for the resource type. The ICON resource contains a single black-and-white icon bitmap, while the ICN# resource contains a list of information on black-and-white and color icons. The ICN# Editor window opens, with a default resource ID of 128. Click on the ICN# item and draw a black-and-white icon design using the editor's drawing tools (see Figure 3.40).

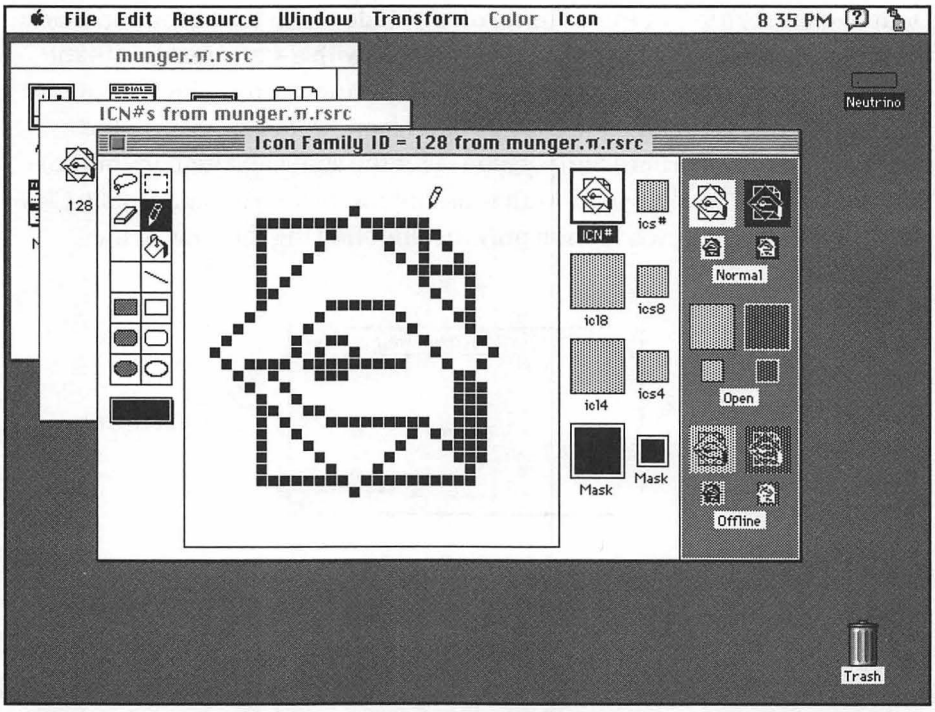


Figure 3.40 Drawing the ICN# resource.

If you look at the icons at the far right, you'll get an idea of how they'll appear on the desktop. They look OK, except for that square outline surrounding it. So the next thing to do is create the icon mask, which is a black silhouette of the icon. The Finder uses the mask data to punch the icon's outline, cookie-cutter fashion, into the desktop background pattern. The Finder then draws the icon into this opening, fitting the icon's image seamlessly onto the screen. Making the mask is easy: Go to the ICN# item at the Editor window's upper right and drag the black-and-white icon down to the mask item window (see Figure 3.41). A silhouette of the icon appears. The appearance of the test display icons should improve dramatically.

Next, drag the ICN# to the icl8 item window and click to select it. Now you can add color to the icon, making an icl8 8-bit color icon resource (see Figure 3.42). Similar to how the dialog editor and dialog item editor work in tandem to produce interrelated resources, the ICN# editor lets you create several types of icon resources. When you're done with the icl8 resource, you can make the icl4 (the four-bit color icon) resource, although it's not necessary. Close the ICN# Editor window and save the file.

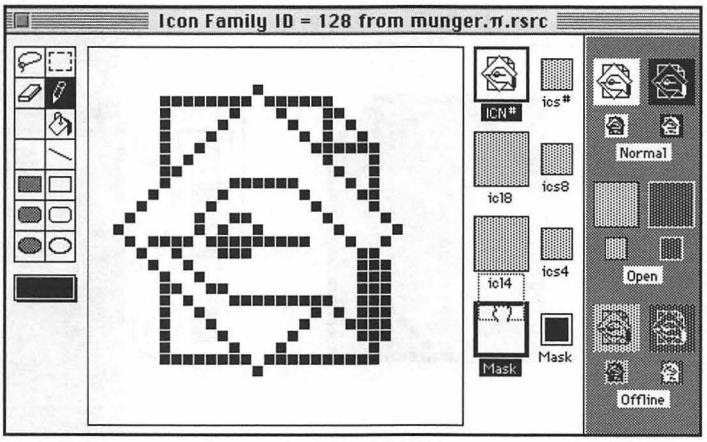


Figure 3.41 Making the application icon's mask.

Select Create New Resource again and the ICN# editor reappears, this time with an ID of 129. Draw a document icon, similar to the one shown in Figure 3.43. When you're done, close the ICN# Editor window and save the file. The ICN# Resource window shows the icons, along with their ID numbers. Close this window and you'll see the various resources associated with the icon list resource.

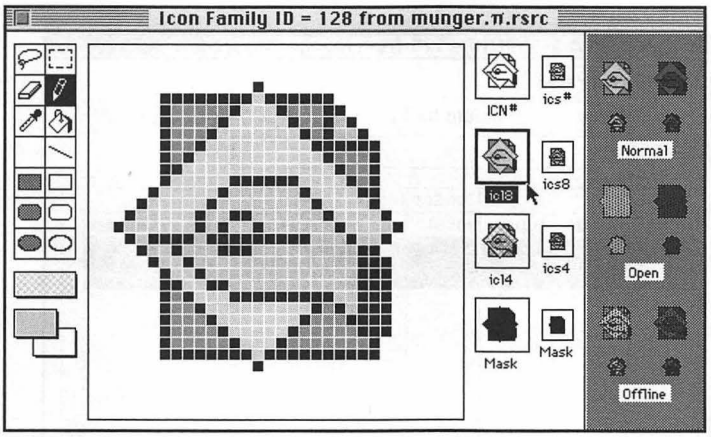


Figure 3.42 Editing an ic18 resource for the application icon.

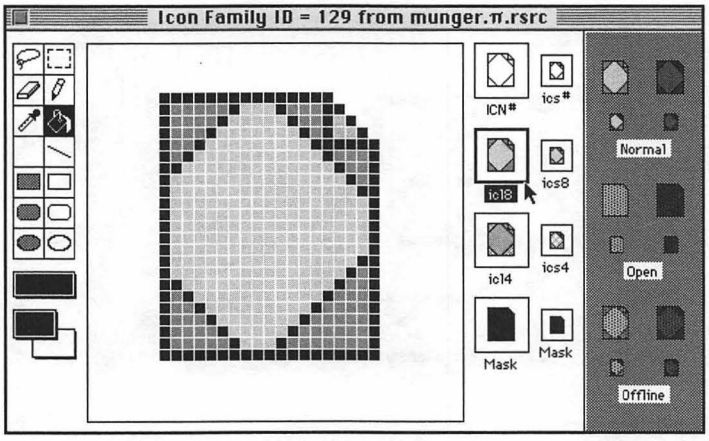


Figure 3.43 Editing an icl8 resource for the document icon.

Select Create New Resource again and this time type **BNDL** in the resource type selection window. The BNDL resource window opens, followed by the bundle Editor window. Go to the BNDL menu and choose Extended View. Type **MUNG** for the signature, to match what you entered in the Project preferences panel in CodeWarrior. Now go to the Resource menu and pick Create New File Type. You'll get a new, highlighted entry in the bundle Editor window, as shown in Figure 3.44.

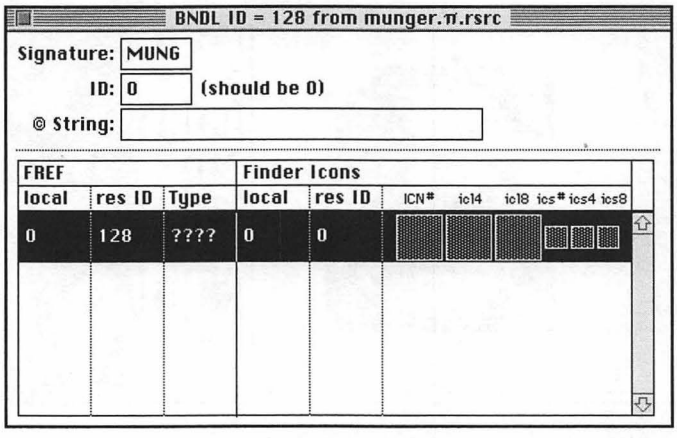


Figure 3.44 Entering a new file type in the BNDL Editor window.

Move the pointer to the Type item, and notice how it changes to an I-beam symbol, for text entry. Type in APPL to replace the four question marks. Next, go over to the icon section and double-click on it. You'll get a dialog box asking you for the icon to use (see Figure 3.45). Click on icon resource 128 and click OK. The empty boxes in this section of the Editor window are filled with icons. Select Create New File Type again and type in TEXT for the Type item.

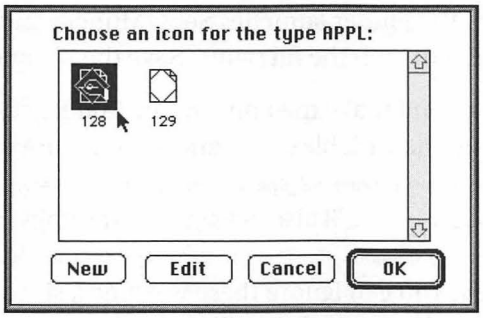


Figure 3.45 Picking the icon for the application file type.

Pick icon ID 129 for the TEXT file type icons. The BNDL Editor window should appear as shown in Figure 3.46.

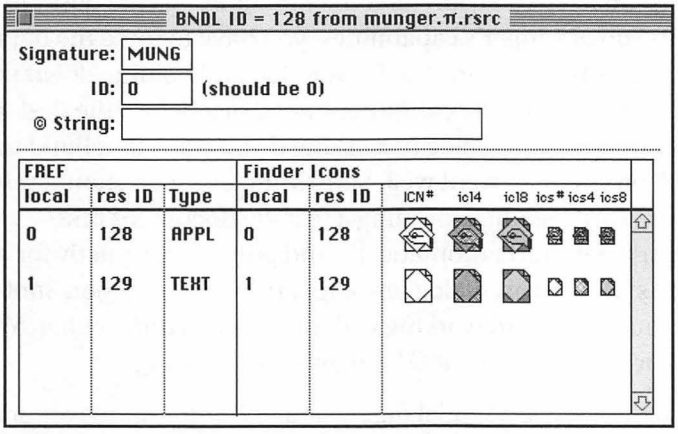


Figure 3.46 The BNDL resource for SonOMunger, with icons for two file types.

Close this Editor window. You'll notice that beside the new BNDL resource is a resource type of 'MUNG'. This is the application's signature resource, made when you typed in those four characters in the signature item of the bundle editor. There's also an FREF, or file reference resource. This resource is used by the Finder to determine what file types (if any) your application recognizes. When you drag and drop a document on SonOMunger, the Finder compares the document's file type to the file types in SonOMunger's FREF resource. If there's a match (say, a ClarisWorks text file was dropped onto SonOMunger), the Finder launches SonOMunger and sends it an Open Documents Apple Event with the filename. Save the file and quit ResEdit.

At long last, go ahead and make the application. You'll get a flurry of warning messages stating that the variables `reply` and `refIn` are not used in the functions `Core_AE_Open_Handler()`, `Core_AE_OpenDoc_Handler()`, `Core_AE_Print_Handler()`, and `Core_AE_Quit_Handler()`. You'll also get similar warnings for `messagein`, except for the function `Core_AE_OpenDoc_Handler()`, where we actually use this variable to receive the file list. You can ignore these warnings, since the variables simply act as placeholders in these handlers.

Finishing Up

The SonOMunger application might still be showing the generic application icon after it is created. To ensure that the Finder brings the desktop database up to date on SonOMunger's capabilities, you have to force the database to be rebuilt. Do this by restarting the Power Mac and holding down the Command and Option keys as the computer boots. Just before the desktop appears, you should get a dialog box asking if you want to rebuild the desktop file. Click on OK. If all went well, SonOMunger's icon should resemble the one we drew in ResEdit. Now drag a text file document onto SonOMunger. It will start automatically and grind away quietly for a few seconds. The status report dialog box appears briefly and then SonOMunger quits. You're not limited to working with one file at a time, either. You can drag several or more files to SonOMunger for processing.

This is why we don't use a modal dialog or alert for the status report, because SonOMunger would stop until you clicked on the OK button for every report displayed. Note SonOMunger lets the user have it both ways for choosing

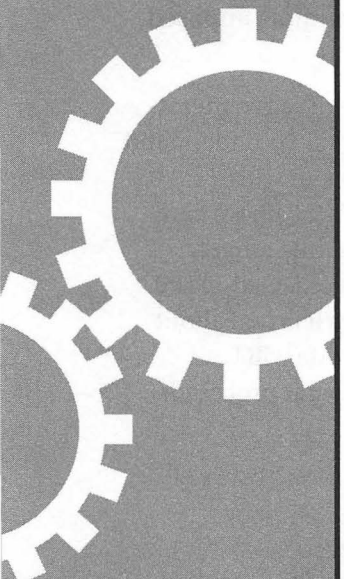
files. The person familiar with the Standard File dialogs can use those to select files, while another person might like the drag-and-drop approach. As you design Mac applications, always remember, give the user as many ways as possible to operate it.

The Fork in the Road

In this chapter you learned about the forked nature of Mac files. We also learned about resources, the building blocks of Mac applications, and how to edit them in ResEdit. We've learned about both low- and high-level events and how to write a Mac application to respond to them. So far, the Power Macintosh looks pretty much like a 680x0 Mac, even when programming it. However, although things look the same, the run-time architecture of the Power Mac is fundamentally different. We'll find out about that in the next chapter.

The PowerPC Software Architecture

The material in this chapter will be of interest to all Macintosh programmers, no matter what their level of expertise. It explains how fundamentally different Power Macs are under the hood, even though they look and behave like 680x0 Macs.



Our road trip has covered quite a bit of ground. We've obtained a nodding acquaintance with the CodeWarrior IDE and learned about the structure of Mac files and applications.

We've made an application that provides a friendly interface and performs useful file I/O. Importantly, this code compiles and runs whether we use the 680x0 CodeWarrior compiler or the PowerPC (PPC) CodeWarrior compiler. While this appears to trivialize the differences between a Mac and Power Mac, make no mistake: The computers use very different processors. Given that fact, the ability to use the same source code to make processor-specific versions of an application is actually a tremendous technical achievement. Apple has put a lot of effort into making the switch to the Power Macintosh as painless as possible. This effort will pay off for users and developers in the following ways: Users' current 680x0 application software is still usable and runs with decent performance. Developers can rapidly port code to the Power Mac without a major effort to produce a native application.

An added plus for both users and developers is a significantly faster application. This improved performance, combined with the low investment in cost and resources to support two different computers, is a win-win situation. It means that you should see lots of native applications appear early on. And in fact, that has been the case: within a year, over several hundred Mac applications were revised as native applications.

While providing compatibility with the past, Apple also engineered the future into the Power Macs. Behind the consistent application interface, the run-time application architecture of the Power Mac has fundamentally changed. It eliminates some of the limitations inherent in the existing operating system design—limitations that arose out of hardware constraints imposed by the 680x0 processor.

This chapter will serve as a rest stop on our journey. While we're recuperating, I'll describe the new run-time architecture in some detail. To understand the new, however, we must first understand the old. Let's begin with a description of the existing 680x0 application architecture. After all, we can anticipate that this type of application will be around for a while longer, thanks to the Power Mac's 680x0 emulator and the millions of 680x0-based Macs in the industry. Finally, remember that everything you learned about the Mac application's program structure still applies: The code will load resources, have an event loop, and call handlers no matter what processor you write for.

The 680x0 Application Run-Time Architecture

As we discovered in the last chapter, Macintosh files are composed of a data and resource fork. For a PowerPC/Power Mac application, the program's code resides in the file's resource fork, as resources of type 'CODE'. Accompanying these CODE resources are other resources, such as DLOG, ALRT, WIND, and MENU, which supply graphical information (such as icons) or data lists (dialog or menu items) that define the application's user interface. A SIZE resource provides operating system information, such as the amount of memory the application needs, whether or not it can run in the background, and if the application is high-level event aware. The data fork of the application is usually empty (see Figure 4.1).

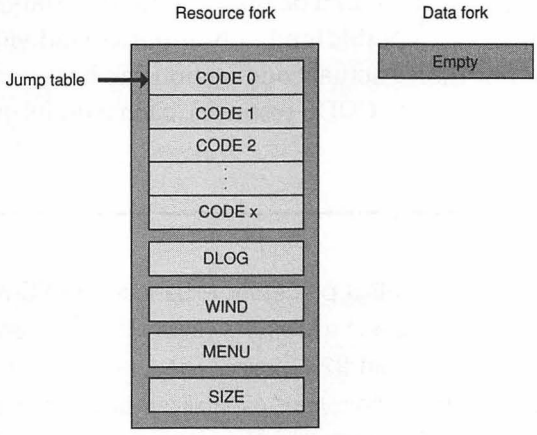


Figure 4.1 The structure of a 680x0 application file.

The application's code section is composed of individual CODE resources, or code segments. Code segments can be a maximum of 32K in size. This value came about due to a limit imposed by the 68000 processor used in the original classic Mac. In order to shoehorn code into the confines of that first system's 128K of RAM, the engineers designed the program code to be position-independent. That is, the code uses no absolute addresses. Due to the shifting memory demands of a running application, code segments could be unloaded and subsequently reloaded into memory in different physical addresses at different times, which is possible only if the code is position-independent. This allowed those early Mac applications to run within the cramped memory space by purging unused code segments and then loading

only those segments that had to execute at the moment. Naturally, in the scheme of the Mac OS design, a Segment Manager deals with these code segments.

The code references (such as a branch to a different part of the program) of such position-independent code are based on the program counter's current address, plus an offset. This scheme is commonly known as PC-relative addressing, the term coming from the abbreviated name of the processor's program counter (PC). The 68000 processor implements PC-relative addressing with a 16-bit signed value, which allows an address range of plus or minus 32K. The offset's sign indicates whether the reference is before or after the current PC address. The trade-off was that while this scheme made the best use of tight memory, it also constrained the code segment's size.

To guarantee that any function within the segment was accesssible to another function, a segment could be no larger than the largest offset possible, or 32K. Remember that this limit only applies to individual CODE resources. The application's actual code section can be rather large, packed with tens or hundreds of 32K CODE resouces, each with its own unique ID number.

Background Info

Later generations of the 680x0 processor expanded the PC-relative offset to a signed 32-bit value, which allows the possibility of generating code segments that are larger than 32K in size. The latest version of CodeWarrior exploits these hardware capabilities by providing several code "models." A code model is a programming strategy that deals with memory addressing issues, such as the implementation of function calls. The several code models used by CodeWarrior are near, far, and smart.

The near model uses 16-bit PC-relative addressing for function references, with all its limitations. The far model uses 32-bit absolute addressing for its function references, thus enabling you to make large code segments. How does CodeWarrior make position-independent code in this situation? The Metrowerks compiler makes all function references into 32-bit offsets from the start of the code segment. When the application launches, glue code obtains the segment's current location in memory and adds this value

to these offsets. The smart model is a combination of the previous two models. For those function references that have already been defined and are within 32K of the calling function, a 16-bit PC-relative offset is used. Otherwise, absolute addresses are used. The smart model thus generates efficient code while sparing you the details of code segmentation.

This brings up a question. Given the 32K PC-relative addressing limit, how does one function call another, especially if the target function is positioned in physical memory beyond this addressing limit? Or, what if that particular CODE resource isn't in memory at all? This problem is dealt with by using a data structure called a jump table. By way of explanation, let's start by reviewing how an application launches.

When you double-click an application icon, the Finder obtains the filename, which it then passes to the Process Manager. The Process Manager examines the application's SIZE resource to determine the size of the memory partition—a contiguous section of physical memory—it must build for the application.

The memory partition subsequently gets divided into three sections. They are referred to as heap, stack, and A5 world (see Figure 4.2). The heap is a data pool that the program draws from as necessary to load more resources, or to process data. This could be more code segments, any needed graphical resources (such as a window or a menu), data structures used by the Toolbox routines, and the program's data. The heap starts at the lowest memory addresses in the partition and expands upwards. The application's stack (not to be confused with the system stack maintained by the OS) holds temporary variables and starts near the highest addresses in the memory partition. It grows downward, toward the heap. Ideally, the top of the heap and top of the stack never collide.

Practically, if an application crashes with a bomb ID of 28, it means the two have met, with disastrous results. The A5 world holds the application's global variables, QuickDraw global variables, and the jump table. The name A5 world comes from the fact that all of these objects are accessed as offsets from an address stored in the 680x0 processor's A5 register. This A5 world is a fixed size and is situated just above the base of the application stack. After these three sections of the application are set up, the Process Manager transfers control to the application's `main()` function.

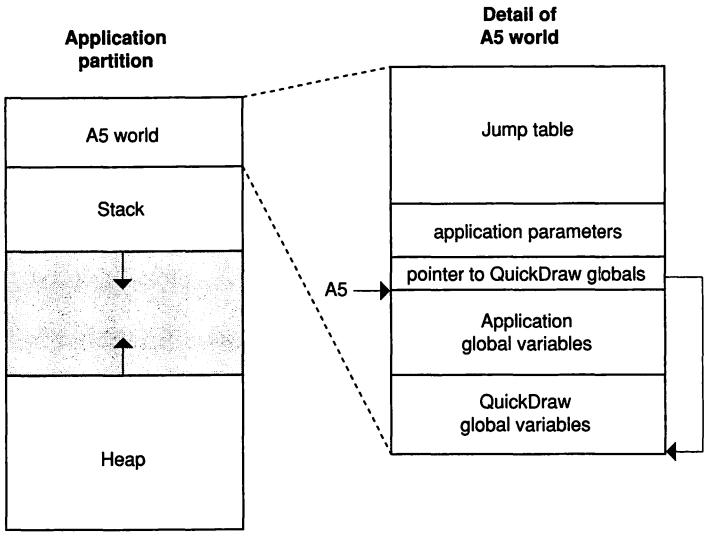


Figure 4.2 The structure of a 680x0 application in memory.

An application's code segment 0 (that is, a CODE resource of ID 0) contains information that the Process Manager uses to set up the A5 world, such as the size of the application's global space and the jump table's initial contents. This segment is built by the development software's linker. When the linker stitches all the program's object code into an application, it keeps track of external function references, that is, calls made to functions outside of a code segment. The linker sorts these references by segment number and then writes this data into the CODE 0 resource. The linker also sizes the global variables used by the application and writes this value into the segment. The Process Manager loads segment 0 into the heap just long enough to establish the A5 world and then discards it. It uses a Segment Manager routine, `LoadSeg()`, to do this.

The final application code produced by the linker has two types of function calls. A function call within a code segment becomes a subroutine jump instruction that uses a PC-relative offset. A call to a function outside the segment becomes a subroutine jump to a jump table entry. Because the jump table is referenced through register A5, this is a subroutine jump instruction that uses the address stored in register A5, plus an offset to a jump table entry. Because the application's global variables must be accessible to every function within the program, they too must be situated in the

A5 world. The application's global variables and QuickDraw globals thus are referenced as offsets from register A5. As you can surmise, tampering with A5's contents is not a good idea, as the application relies on it to both operate and to locate global variables.

Now let's see how the jump table completes the connection to the external function. The jump table is made up of an array of 8-byte entries, as shown in Figure 4.3, where each entry represents a function reference. These entries can have one of two formats. The first format is used when a particular function's segment is already loaded into memory. The corresponding jump table entry contains a segment number (2 bytes) and a jump instruction (2 bytes) with a 32-bit absolute address (4 bytes). Therefore, when an external function gets called, the A5-relative subroutine jump hops to a corresponding entry in the jump table, which in turn becomes a jump instruction to the actual function.

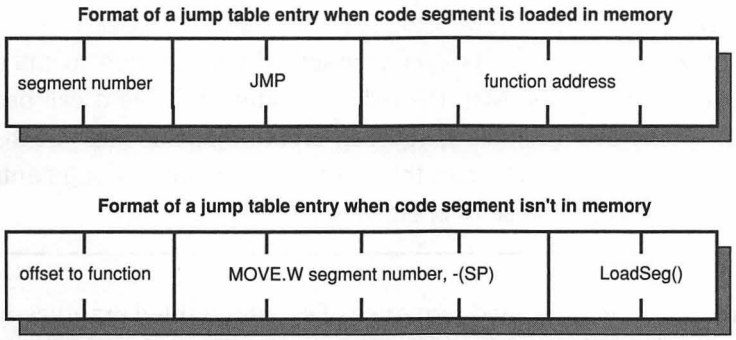


Figure 4.3 The two formats of a jump table entry.

If the segment isn't in memory, the jump table entry uses the second format. The entry contains the target function's offset into the missing segment (2 bytes), followed by an instruction that pushes the segment number onto the stack (2 bytes), and finally a call to the `LoadSeg()` routine (2 bytes). Now the subroutine jump into the jump table executes the push instruction, and then calls `LoadSeg()`. `LoadSeg()` finds the target segment number on the stack, loads the CODE resource with that ID into memory, and locks it there. Next, it takes the offset value in the jump table entry and adds it to the segment's current address in memory to obtain an absolute address for the function's entry point. Remember that the segment might get loaded into different sections of memory, so this absolute address changes each time the segment

is loaded. `LoadSeg()` then converts the jump table entry into the first format so that it now holds the segment number and a jump instruction. It also updates the jump table entries for every function contained in this segment. `LoadSeg()` finally executes the jump instruction it built in the jump table entry, transferring control to the target function. If a segment happens to get purged from memory (via another Segment Manager routine called `UnloadSeg()`), the appropriate jump table entries are revised to the second format to reflect this fact.

As you can see, these operations are transparent to the programmer. The jump table mechanism quietly ensures that when a function is called, if its code segment isn't in memory, it gets loaded there automatically. Releasing memory isn't as automatic: the programmer has to call `UnloadSeg()` to indicate to the Mac OS which segments aren't in use.

Hazard

It's not wise to call `UnloadSeg()` yourself. If you happen to purge a code segment that the Mac OS will need later (such as a call back into an Apple Event handler), you can create a spectacular crash. Instead, let the Mac OS call this routine as it purges segments during the normal course of operations.

This carefully choreographed sequence of events enabled graphics-intensive Macintosh applications to run in small amounts of memory. MultiFinder, Apple's first implementation of cooperative multitasking, was possible because each application's position-independent code and jump table allowed them to be loaded and executed anywhere in memory.

However, there's still a problem: How does an application access Toolbox routines? Most of these routines are in the ROMs, which are located in the Mac's memory space, well over a PC-relative jump away. We didn't see anything in the application's jump table to deal with Toolbox routines.

For the answer, we again turn to the 680x0 processor. Normally, a processor trundles along, fetching program instructions and executing them. Occasionally, the processor might detect a *trap* or *exception* condition. This is an abnormal state that might be caused by the instruction itself (such as a divide by 0, an invalid instruction, or a code reference to an odd address), a

bus error (a memory SIMM or other hardware component failed to respond to a bus access), or a peripheral device requesting service through an interrupt. The processor responds to an exception by first pushing the address of the next program instruction onto the stack, followed by some information—called an exception frame—that's a snapshot of the processor's internal state. The processor then fetches an address from a preprogrammed location in memory whose location is determined by the type of exception that occurred. The processor jumps to this address, which is the entry point to a function that handles the exception. The handler code remedies the problem (if possible), or services the device request. When the handler code completes, the processor retrieves the exception frame from the stack, thus restoring its internal state. Finally, the saved program address is popped from the stack into the PC, which places the processor at the next instruction in the program, no worse for wear.

Motorola defined two special unimplemented instructions for the purpose of extending the capabilities of the 680x0 processor. When the processor traps on one of these instructions, it executes handler code that emulates new instructions. One of these unimplemented instructions is called the A trap word, so called because it's16 bits in length and the first four bits in the word are the bit pattern for the hexadecimal A.

Important
In most of the Apple literature, a word is 16 bits in length. This follows a convention where the size of 680x0 processor's instructions were this length. The current PowerPC processor literature from IBM and Motorola define a word as being 32 bits long. Further complicating this situation is that the PowerPC 620 is a 64-bit processor. Needless to say, this can cause some confusion. For this discussion, we'll stick with the 16-bit word length and keep the use of the term word to a minimum.

Apple used the A trap as an entry point into its Toolbox routines. In its header files, each routine is assigned a word that starts with hexadecimal A, followed by bits that indicate the routine type, some flag bits, and an 8- or 9-bit value. For example, if we peek at the "Dialogs.h" header file, and search for the `StopAlert()` routine, we find the macro:

```
extern pascal short StopAlert(short alertID, ModalFilterUPP modalFilter)
  ONEWORDINLINE(0xA986);
```

The macro ONEWORDINLINE reduces the declaration to:

```
extern pascal short StopAlert(short alertID, ModalFilterUPP modalFilter)
➥\ = {0xA986};
```

Here we see that the two arguments, `alertID` and `modalFilter`, will be pushed onto the stack, using the Pascal language calling convention. This is followed by the trap word for the `StopAlert` routine, 0xA986. Every Toolbox routine uses similar macros that place arguments on the stack or in certain registers, and then hands the job off to the exception handler. If you disassemble a 680x0 program using the Disassemble command in CodeWarrior's Project menu, you'll notice the program's 680x0 machine code is peppered with these A trap words, all which correspond to Mac Toolbox calls.

Background Info

The last two bytes of jump table entries for code segments not loaded in memory (the second format) are a call to the `LoadSeg()` routine. These bytes contain the trap word 0xA9F0, which is the `LoadSeg()` trap.

Let's put this all together. A Mac is running an application with the 680x0 processor dutifully fetching and executing instructions. Suppose the program now calls a Toolbox routine. When the processor hits the A trap word that represents this routine, it causes an exception. The processor fetches the address for the location of an A trap exception handler written by Apple and executes it. This handler—appropriately called the Trap Dispatcher—examines the trap word and uses the type bit to select one of two dispatch tables. One table is for the low-level routines, the other is for operating system routines.

The Trap Dispatcher then uses the trap word's lower 8 or 9 bits to calculate an offset into the particular dispatch table. The entry at this offset in the dispatch table contains the address of the Toolbox routine. Typically, this is an address in ROM, but some routines can be found in RAM. The processor hops to this address and executes the Toolbox routine. When the routine completes, the processor returns from the exception, back to the next instruction in the application. Where do the addresses in the dispatch table come from? They're stored in the Macintosh's ROM and are loaded into the dispatch table when the Mac starts.

Using the exception mechanism as an access point into the Toolbox seems a tad complicated, but the design has some important advantages. First, it allows a code segment anywhere in memory to readily access the Toolbox routines. Second, this mechanism provides flexibility to fix bugs or add new services. For example, assume that it's discovered that the Toolbox routine `ReallySuperbService()`, located in ROM, has a bug. We know that you can't change ROM—but you can change the offending routine's address in the dispatch table. Built into the Mac's boot process is a procedure for installing patch code. After the dispatch table is built, but before initialization is completed, the System file (early Macs) and System Enabler files (current Macs) are searched for patch code resources. These resources are loaded in memory, locked, and executed. This code modifies the address for `ReallySuperbService()` in the dispatch table so that it points to the improved version of the routine located in RAM, rather than the one in ROM.

Apple uses the same method to add enhancements or new services to the Mac OS. The code implementing new features is loaded and locked in memory. Empty entries in the dispatch table are directed toward routines within the feature code. Apple is thus able to fix bugs or add features to the operating system of existing Macs with just a new release of the system software.

Third-party vendors can also supply enhancements through the use of Extension files and Control Panel files. These files have INIT resources that contain the enhancement code, plus code to patch the dispatch table. At boot time, the operating system first installs any patch code; then it searches the Extensions folder and Control Panels folder for files, installs their INIT code, and modifies the dispatch table. Apple's own CD-ROM driver, QuickTime software, and File Sharing software are installed this way.

Future Directions

The Mac OS is heavily dependent upon Toolbox routines stored in the ROMs, as I have mentioned previously. This makes the Mac virtually impossible to clone because laws have established the copyright value of ROM code. (Ironically, these laws came about when Apple sued another vendor who was cloning Apple II computers.) With Copland, this state of affairs changes. Copland will no longer be a ROM-centric OS.

continues

continued

Instead, its Toolbox and OS routines are groups of shared libraries stored as disk files. This simplifies updating or enhancing Copland, because upgrades are accomplished by just adding new versions of the system files to your computer's hard drive. This also makes for easy distribution of the Mac OS to licensed Mac clone vendors such as Radius, DayStar Digital, and Power Computing. The downside to this design is that you'll need a bigger hard drive to hold the Copland OS, but this is hardly an issue since 500M and gigabyte drives are a common staple these days.

The PowerPC Application Run-Time Architecture

On the surface, a PowerPC Mac application seems identical to its 680x0 counterpart. As mentioned earlier, the code you wrote in Chapter 3 compiles and runs on a Mac with either processor. However, the run-time architecture behind the API is fundamentally different.

We can see a difference immediately when we examine the structure of a PowerPC Mac application. Looking at Figure 4.4, you can see that that application's resource fork still has the graphical resources and the SIZE resource. However, the program code is located in the file's data fork, as a block of PowerPC code known as a *code fragment.* This code fragment isn't segmented, nor is there a size limit. Thus, all of a PowerPC application's code is stored in a single code fragment.

An application with 3M of PowerPC code has a code fragment 3M in size in the data fork, plus whatever resources are required in the resource fork to implement the user interface. After viewing the gymnastics required to support 32K segments in a 680x0 Mac application, the PowerPC application design appears starkly simple.

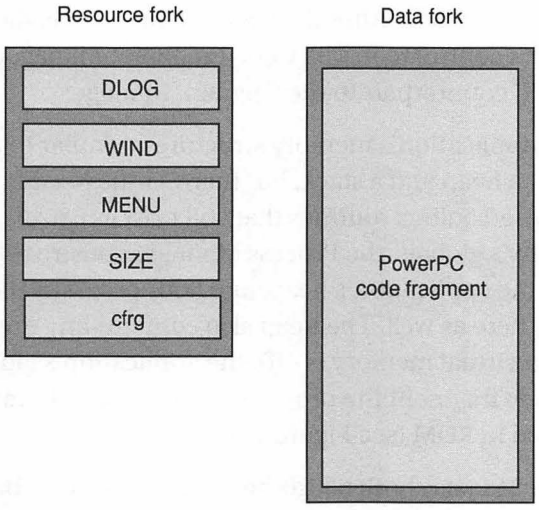


Figure 4.4 The structure of a PowerPC application file.

Background Info

Lest you think the original Mac design team came up with an unnecessarily complex design, remember that they were working in an era when 256K to 512K of RAM was considered adequate, and that the 68000 processor could only address a maximum of 4M. The simplicity of the PowerPC software architecture stems from the capabilities of today's hardware. The basic Power Macintosh configuration has 8M of RAM and supports virtual memory. The PowerPC 601 processor in these first Power Macs can address 32 bits of physical memory (4G) and 52 bits of virtual memory (4T).

Finally, the Power Mac's System Software engineers had the advantage of a decade's worth of improvements in operating system technology.

Launching a PowerPC Mac application is similar to that of a 680x0 Mac application, up to a certain point. When you double-click the application icon, the Finder gets the filename and passes it to the Process Manager as before. However, now the Process Manager calls a Code Fragment Manager, whose job is to load code fragments into memory, lock them there, and

prepare them for execution. After the code fragment is readied, the Process Manager transfers control to it. The Code Fragment Manager can be considered the PowerPC counterpart to the Segment Manager.

The Power Mac application's memory structure is similar to a 680x0 application. There's still a heap and a stack, but there's little need for an A5 world. However, for those Toolbox routines that still exist as 680x0 code and need to access QuickDraw's globals, the Process Manager constructs a pointer to these globals in the application's heap, and your program allocates storage for these globals here as well. The heap also contains any executing code fragments (when virtual memory is off), the application's globals, the globals of any library code fragment the program uses, and any library code fragments not located in ROM (see Figure 4.5).

Because a Power Mac has both 680x0-based and PowerPC-based Mac applications on it, how does the Process Manager know which Manager to use when you launch an application? Each PowerPC application gives the Process Manager a hint: They have a resource of type 'cfrg' in the file's resource fork. This resource tells the Process Manager that this application contains PowerPC code, so it uses the Code Fragment Manager to load the application. If the cfrg resource is absent, the Process Manager assumes the application is a 680x0 binary and calls the Segment Manager instead. The cfrg resource is placed in the file by the development software.

What about the Toolbox routines in ROM? They, too, are code fragments. After the Code Fragment Manager loads the application's code fragment into memory, it goes about resolving any external references, which are usually the Toolbox calls. The Code Fragment Manager loads any additional code fragments into memory (recall that not all Toolbox or operating system routines are in ROM), and then it replaces each external routine reference with its actual address.

To see how this is done, let's examine code fragments in more detail. Code fragments come in two executable formats, XCOFF and PEF. XCOFF is the acronym for IBM's Extended Common Object File Format, whereas PEF stands for Apple's Preferred Executable Format. As its name implies, the preferred format for code fragments for the Power Macintosh is the PEF layout. XCOFF is partially supported because the original IBM development tools used this format.

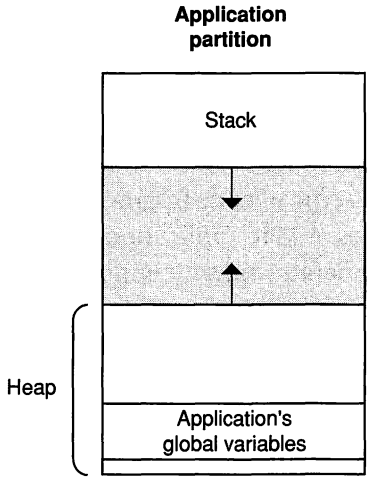


Figure 4.5 The structure of a PowerPC application in memory.

Important
The Code Fragment Manager uses a set of routines known as the Code Fragment Loader to load code fragments from a file into memory. The Code Fragment Loader's function is analogous to `LoadSeg()`'s. The Code Fragment Loader is responsible for recognizing and loading either XCOFF or PEF files. If new file formats are introduced later, the loader will be updated to handle them. You, the programmer, needn't concern yourself with file formats. Just let the Code Fragment Manager handle the job of loading your code fragments.

A PEF consists of a container of code, data, and loader information block. A container is a chunk of contiguous storage, typically a file, although it can be any object that the Mac OS accesses, such as the libraries that house the Toolbox routines in the Power Mac's ROMs. The code and data make up the code fragment itself, and the loader information block enables the Code Fragment Manager to prepare the fragment for execution. The loader information describes the fragment's initialization, start, and termination functions, its imported functions and data, its exported functions and data, and its version number.

The import/export information is crucial to the operation of the PowerPC run-time architecture. It's how the connections between an application and the Toolbox routines are established. Code fragments can export certain entry points or import the entry points of data objects or functions from other code fragments. For example, the Mac Toolbox is a shared library in the Power Mac's ROMs. This type of code fragment exports the entry points of its global data and routines. A Mac application, on the other hand, requires Toolbox routines to operate and so it imports the required entry points from the shared library in the ROMs. The development software's linker is responsible for matching up the import names in the application to export names in a shared library. The linker places the exporting library's name and any import names into the code fragment's loader information block. It's important to note that this information is stored as actual name strings. These names get resolved to addresses by the Code Fragment Manager at run time.

Background Info

It's easy to see what libraries and routine names a code fragment requires. To do this, make a copy of the SonOMunger application. Now launch ResEdit. In ResEdit's File menu, select Get File/Folder Info and open the copy of SonOMunger. In the Info box that appears, change the Type item from APPL to TEXT, close the window, and save the file when ResEdit asks you to. Quit ResEdit. Now, open this file with any word processor. You'll see some gobbledygook—that's binary machine code, but there's also a block of text that you can easily read. This block begins with the library name "InterfaceLib," followed by the name of every Toolbox routine used by the application.

When Code Fragment Manager loads the application's code fragment, it first allocates memory for the global variables and static data in the heap space of the partition built by the Process Manager. The Code Fragment Manager then performs any load time relocations for the import symbol information and places this information in a critical data structure called the table of contents, or TOC. The TOC was built by the development tool's compiler and

linker, and it contains the fragment's import symbols (that is, the names of the externally referenced data or functions).

The Code Fragment Manager resolves these import symbols and plugs addresses in the appropriate slots in the TOC. The TOC contains lists of three type of pointers. These pointers reference the code fragment's own functions, its own data, and the import names it uses. These import name references are the global data variables or the entry points of functions in other code fragments.

To set up the addresses in the application fragment's TOC, the Code Fragment Manager uses the library names in the loader information block to locate the required shared libraries. It loads these libraries into memory, if required, and loads any other libraries that these libraries depend on. The Code Fragment Manager also runs each library's initialization function code, if present. The shared libraries build any data structures they use within the application's heap, and some of the TOC pointers are arranged to point at this data. The Code Fragment Manager then searches for the application code fragment's import names and replaces them with the corresponding export addresses in the shared library, in a process called *binding*. This binding operation sets up the remaining TOC pointers (see Figure 4.6). After the TOC is initialized, the code fragment's preparation is complete, and it is ready to execute.

Note that some of these TOC pointers address objects called *transition vectors*. A transition vector is a data structure used by one code fragment to access an import function in another code fragment. The structure consists of one pointer to the target fragment's TOC, and a second pointer to a function within the target code fragment. Therefore, a shared library doesn't actually export the addresses of its routines. It instead exports transition vectors, whose job is to point to the routines. The transition vectors are built by the development software.

Because the TOC is the linchpin of the code fragment's operation, one of the PowerPC processor's general-purpose registers (GPR2) points to the start of the TOC at all times and is called the TOC Register (RTOC). The RTOC serves a function similar to register A5. However, whereas only 680x0 applications could have an A5 world, any PowerPC code fragment (a plug-in module, extension code, or a driver) can have a TOC.

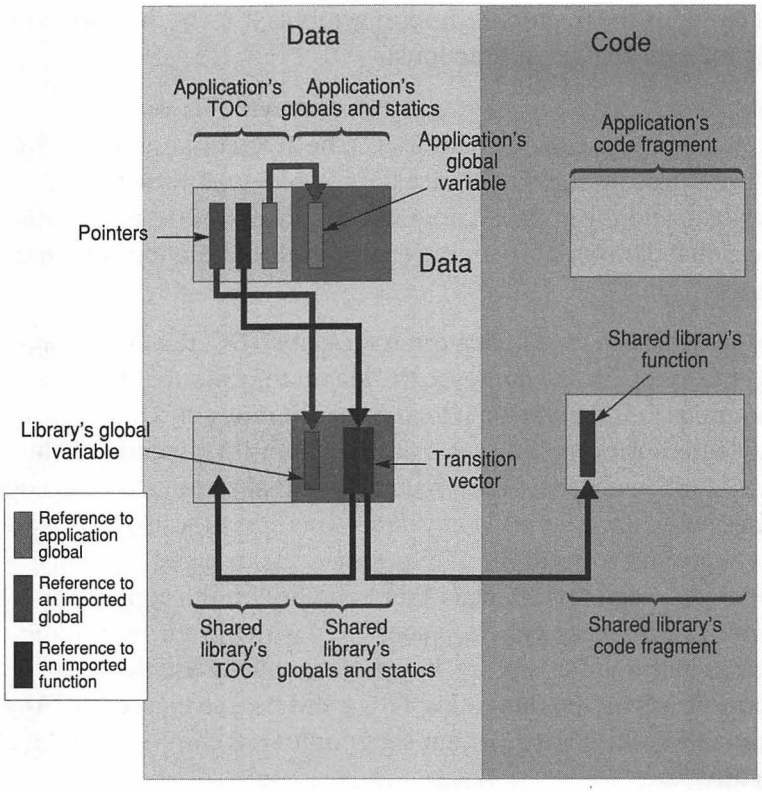


Figure 4.6 The run-time binding of the application code fragment to its libraries.

The dynamic linking strategy used by the Code Fragment Manager minimizes the copies of shared libraries in memory, especially in a multitasking environment. As you've just seen, each application that uses a library has its own instance of the library's data built for it, unless the library implements a special shared memory strategy. Because the library's code is separate from its data, each application can thus execute the same library code while using its private copy of the data. The shared libraries remain in memory as long as any application uses them. If the library isn't being used, its termination code (if any) is executed, and it's unloaded from memory.

Background Info

Because each application can have its own data copy while using a shared library routine, this capability is said to be *reentrant* (that is, usable by multiple processes simultaneously without conflicts). Thus, the Power

Macintosh's Toolbox routines are a major step toward the day when Apple releases Copland, which uses preemptive multitasking. It's important to note that this method of data storage doesn't guarantee that a program's code is reentrant. However, unless a code fragment library can maintain private data storage for each process that uses its library routines, reentrant code isn't possible.

Whereas a 680x0 application's global variables are intimately tied to its A5 world, a PowerPC code fragment's global variables are readily accessible to other code fragments through its TOC. This makes it easier to access and share global data than was possible with the 680x0 run-time architecture. Previously, periodic tasks, extension code, or plug-in modules had to use assembly language code to gain access to the global variables within an application or inside the operating system. With the PowerPC run-time architecture, no special programming is necessary to obtain access to information within another code fragment.

We've covered how data can be accessed by different code fragments. To complete our understanding of the run-time architecture, let's consider how one code fragment function calls a function in another code fragment. Suppose that a code fragment, our Power Mac application, makes a Toolbox call. The imported function address is fetched from the appropriate transition vector and execution hops to the Toolbox code fragment. However, an executing code fragment assumes that the RTOC points to its own TOC, which contains its globals, and addresses of any import functions in another code fragment. How is the RTOC set to this new code fragment's TOC?

The run-time architecture assigns this job to the caller. In other words, before execution passes another code fragment, the program must set the RTOC to point to the target code fragment's TOC. This information is stored in the transition vector.

Getting back to our example, the following three events occur when our application makes a Toolbox call. First, glue code in the application uses the transition vector to set the RTOC to the TOC of the Toolbox's shared library. The glue code then uses the other half of the transition vector to jump to the Toolbox routine. Finally, when the routine completes, execution returns to the application code fragment, and the RTOC is restored to the application's TOC.

Background Info

Following the RISC principle of a simple instruction set, the PowerPC processor has no "call subroutine" or "return from subroutine" instructions. Subroutine "calls" are implemented as branch instructions surrounded by additional instructions to set up registers for function arguments and to preserve critical registers. Subroutine "returns" are typically branch instructions that use an address stored in the processor's link register (LR).

As an example of this, let's look at the machine code for calling a Toolbox trap, `WaitNextEvent()`. In 680x0 machine code this is

```
WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR)


MOVE.W    #$FFFF, -(A7)  /* Load the everyEvent mask onto stack */
PEA       $FFA8(A5)      /* Push address of global gmyEvent
                            onto stack (note A5 reference) */
PEA       $00C3          /* Push LONG_NAP (decimal 60) */
CLR.L     -(A7)          /* Push NO_CURSOR */
WaitNextEvent            /* Trap word 0xA860, go directly to
                            the Trap Dispatcher */
```

Notice that the arguments are pushed on the stack (register A7), and that a trap word takes the processor to the exception handler, the Trap Dispatcher. For the PowerPC, this same function call becomes:

```
addi r31, RTOC, 648  /* Put address of global gmyEvent into r31 */
     .
     .               /* Other program code */
     .
li   r3, -1          /* Load the everyEvent mask */
mr   r4, r31         /* Get the address of gmyEvent */
li   r5, 60          /* Load LONG_NAP */
li   r6, 0           /* Load NO_CURSOR */
bl   .WaitNextEvent  /* Branch to WaitNextEvent(), save return
                        address in link register */
lwz  RTOC, 20(SP)    /* Fix up RTOC to point back to app's TOC */
```

Here the arguments get placed into registers and then a branch is taken into the glue code responsible for managing the jump to the Toolbox shared library. This branch instruction also saves the program's next

instruction address into the LR, providing a way home when `WaitNextEvent()` returns. The glue code, meanwhile, loads the pointer to `WaitNextEvent()`'s transition vector from your application's TOC. This glue code uses the transition vector information to adjust RTOC to the TOC of the Toolbox's shared library and then the jump to the shared library occurs. When the routine returns, the RTOC is immediately set back to our code fragment.

Where does this glue code that accomplishes this magic come from? It's in Interface.Lib.

Unfortunately, this elegant scheme is complicated by the fact that not all of the Toolbox code in the Power Macs is PowerPC code. Rewriting the Mac Toolbox, which consists of nearly 2M of tight CISC processor code (based on the size of the 680x0 code inside the Quadra 840AV's ROMs) into RISC code was a formidable process at best. Not only was the job a large one, but replacing time-proven routines with new ones opens the door to introducing bugs. To achieve high compatibility with 680x0 applications and still get the Power Macs into the hands of users as soon as possible, Apple elected to rewrite only a portion of the Toolbox. The remaining routines were left as 680x0 code and the 68LC040 emulator executes them. However, as Apple extends the Mac OS, the system enhancements are written as native code. For example, in the second-generation Power Macs, the Expansion Manager (used to handle the PCI bus) and the network protocol stacks (used to implement Open Transport, the new network interface) are native code. Copland will complete this transition of the Toolbox from 680x0 to PowerPC code, since it's supposed to be largely native code.

Segue: The Care and Feeding of Stack Frames

So far this discussion has evaded describing how the PowerPC processor maintains two separate code environments. To understand how this is done, this section covers this mechanism in some detail. Typical programmers won't be concerned with how the system operates at this level—nor should

they be. However, if you're curious as to how this happens, or expect to be spelunking through your program code in a low-level debugger, your time reading this section is well spent.

680x0 Function Calls

Like the run-time architecture description, we'll start with how the 680x0 processor handles function calls. It uses a stack pointer (SP) that indicates the position of the stack's top in memory. On the 680x0 processor, register A7 serves as the SP. Processor instructions that push data onto the stack (such as a function's parameters) or pop data off the stack (perhaps a function result) accomplish this by reading and modifying the SP's value. To avoid bus errors, the 680x0 processor ensures16-bit alignment of the parameters it pushes onto the stack.

During the life of a function's execution, it needs storage for the parameters passed to it, the return address, and any local variables it uses. Typically, this temporary storage is obtained from the stack. That region of the stack dedicated to the current function's storage needs is called a *stack frame.* A *frame pointer* (FP) points to the base of the current function's stack frame in memory. The Mac OS uses register A6 as the FP. Figure 4.7 shows the 680x0 stack after a function has been called.

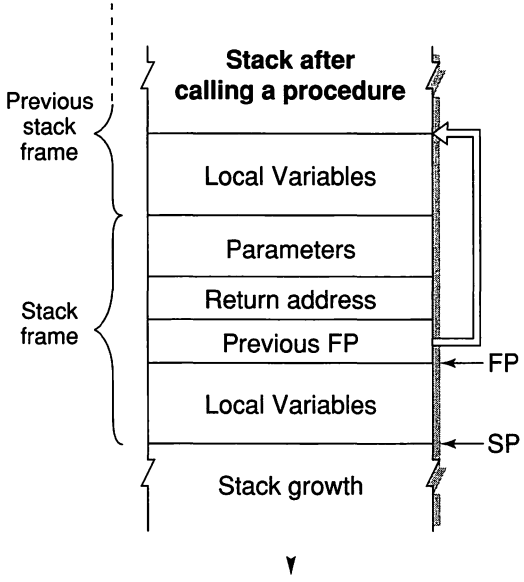


Figure 4.7 The 680x0 processor stack with a stack frame.

The FP serves two purposes. First, a function's parameters are placed on the stack above the FP, and its local variables are placed onto the stack below the FP. This simplifies code design because you use a positive offset from the FP to access function parameters, whereas a negative offset lets you access any local variables. Second, because the Mac OS manages the FP, this makes it easy to locate the start of a function's stack frame. When a function exits, the Mac OS just discards its storage by bumping the SP to the top of the stack frame.

A C function's parameters are pushed onto the stack from right to left. The right-to-left convention enables C functions to support a variable-length function parameter list. This arrangement allows for program code flexibility, but also contributes some run-time overhead. Because only the calling function knows how many parameters it pushed, it must also fix up the stack (by popping the parameters off it) when the called function exits.

Function results (if any) get returned in the 680x0 processor's D0 register. For those processors equipped with a Floating-Point Unit (FPU), floating-point results are returned in the FPR0 register. Table 4.1 summarizes the Mac OS's use of the 680x0 processor registers.

Table 4.1 680x0 Processor Register Usage

Register	*Purpose*	*Preserve Contents across Function Call?*
A0	General use	No
A2	General use	No
A3	General use	Yes
A4	General use, Alternate globals pointer	Yes
A5	Globals pointer	Dedicated, do not modify
A6	FP, Alternate globals pointer	Yes
A7	SP	Dedicated, do not modify
D0	General use	No

continues

Table 4.1 Continued

Register	*Purpose*	*Preserve Contents across Function Call?*
D2	General use	No
D3	General use	Yes
D7	General use	Yes

We've used C language conventions in our description of the function's stack frame. Pascal language programs handle function calls differently. First, the calling function pushes space for the called function's result onto the stack. Next, the called function's parameter list is pushed onto the stack in left to right order. Therefore, a Pascal function's parameter list is always a fixed length. The advantage to this scheme is that the size of the called function's stack frame gets determined at compile time, not at run-time as is the case with the C functions.

Important
Because most of the Mac Toolbox started life as Pascal code, you often must pay attention to these Pascal calling conventions, particularly when accessing Toolbox routines at a low level. Also, while we're discussing items on the stack, it's important to note that the OS Toolbox routines don't normally use the stack. Instead, these Toolbox routines pass parameters and return results through specific processor registers.

PowerPC Function Calls

There are several major differences how function calls—and consequently, a function's stack frame—are handled on the PowerPC processor. First, native functions now use a standard calling convention. This provides a consistent access mechanism that all programming languages use. No extraneous glue code is necessary to massage the parameter list, as often happens on the 680x0 processor when a C language function accesses Pascal-oriented Toolbox routines. Second, a PowerPC's stack frame structure is precisely defined. This lets a stack frame's size be determined at compile time, which eliminates the run-time overhead that a dynamically sized stack frame

incurs. Because of the reduced overhead, frequent use of the native Toolbox calls doesn't exact a performance hit.

Finally, while 680x0 calling conventions makes extensive use of the stack for parameter passing and local storage, the PowerPC calling convention is heavily register-based. Because the PowerPC processsor has 32 general purpose registers (GPRs) and 32 floating-point registers (FPRs), this makes sense. In reality, only eight of the GPRs and fourteen of the FPRs are actually used to hold parameters because the other registers are used to maintain the run-time environment. However, eight GPRs should ensure that a function's parameters get passed via the registers most of the time. Table 4.2 summarizes the use of the PowerPC processor registers.

Table 4.2 PowerPC GPR Usage

GPR	*Purpose*	*Preserve Contents across Function Call?*
r0	Linkage/glue	No
r1	SP	Dedicated, do not modify
r2	RTOC	Dedicated, do not modify
r3	First function parameter	No
r10	Eighth function parameter	No
r11	Environment pointer	No
r12	Global linkage use	No
r13	General use	Yes
r31	General use	Yes

As the table indicates, the PowerPC uses register r1 as the SP, and it preserves 8-byte alignment when items are pushed onto the stack. There is no corresponding FP, but the strict conventions regarding a function's stack frame structure makes it unnecessary. Notice that registers r0 through r12 act as temporary, or scratchpad storage, for the function's operations. The contents of these registers are termed *volatile*. If the function modifies the contents of r13 through r31, their values must be restored. These are the *non-volatile* registers.

Figure 4.8 shows the organization of the PowerPC stack with a stack frame.

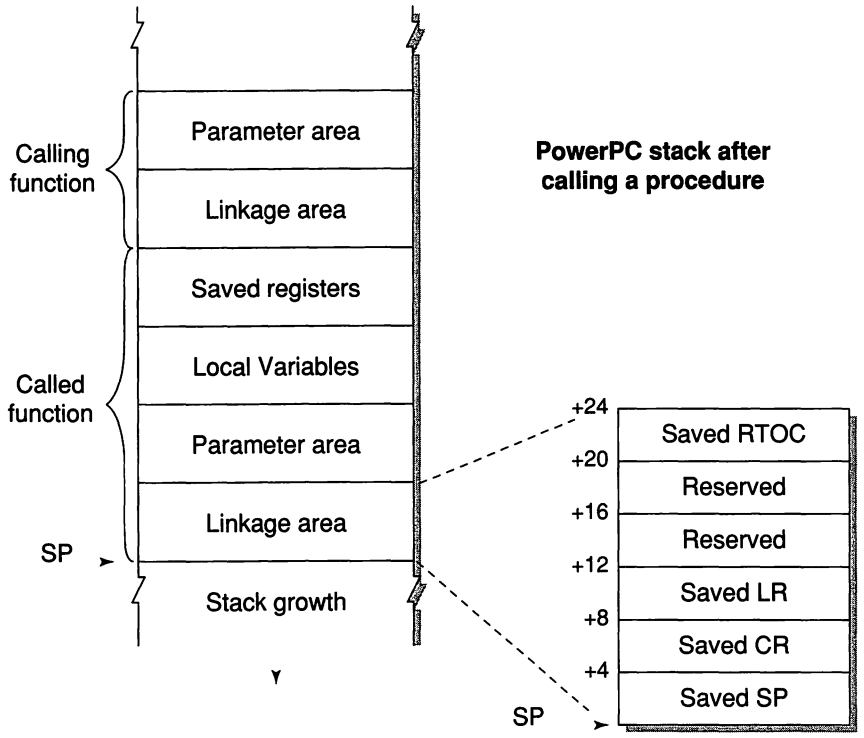


Figure 4.8 The PowerPC stack after a function call.

When a function is called, its parameters are evaluated in left to right order. This arrangement is similar to the conventions used by the Pascal language and guarantees a fixed stack frame size. General or fixed-point values get placed into r3 through r13. Floating-point values get placed into fpr1 through fpr13. Function results are returned via r3 or fpr1. In the case of a handling a data structure (a typical Mac situation), a pointer to the structure is returned.

Those parameters that can't be placed in the first 8 GPR registers are passed through a dedicated area on the stack frame. Furthermore, this space is allocated such that it can hold all of the called function's parameters, even those that do get passed in the registers.

This latter requirement accomplishes two things, especially because the stack frame is a predetermined, fixed size. It first makes some extra copies of the parameters available in the stack frame, which the called function's code can use to do some performance optimizations. Next, this arrangement happens to handle that C function with a variable-length parameter list

because all of its parameters get passed via the stack frame. In this situation the called function retrieves the parameters out of RAM, rather than from the registers.

The stack frame is composed of four regions: a saved registers area, a local storage area, a parameter (or argument) area, and a linkage area. The saved registers area is used to preserve the contents of any nonvolatile GPR or FPR used by the function (that is, r13 through r31, and fpr14 through fpr31). If the function is simple enough so that its code uses only the volatile registers, this area doesn't exist. The local storage area holds the function's local variables, and is a fixed size. The nonvolatile registers are used for local storage first. Additional local variables that can't fit into the registers spill over into the local storage area on the stack. Like the saved registers area, it's possible that a simple function might not have a local storage area. A function uses the parameter area to set up and forward parameters to other functions that it calls. The calling function must arrange this area's contents before every function call. Because a function typically calls many other functions, the size of its parameter area size must be set to accomodate the function that has the largest parameter list.

The linkage area stores the previous SP (which points to the base of the last stack frame), the RTOC, the Condition Register (CR), and the Link Register (LR). With the link area juxtaposed to the stack's top, the calling function can readily access information in this section, and the called function can locate the caller's parameter area.

Let's see how the setup of the stack frame eliminates the need for a frame pointer. When a function gets called, a special piece of code produced by the compiler called a prolog builds the stack frame. It first preserves LR, CR, and RTOC in the calling function's link area. This is done simply using offsets from the SP's current position. The LR contains the program's next instruction address, whose value serves as a return address later. Depending upon the situation, the prolog might not preserve the CR or RTOC registers. For example, a function inside your own program's code fragment uses the same TOC, and so the prolog code doesn't save the RTOC's contents. This eliminates the overhead of two move instructions from the function call.

Next, the contents are written into the save area. Finally, the prolog code allocates space for the stack frame by decrementing the stack pointer, and then it writes the previous SP into its own link area. This value is occasionally called the *back chain* because it points to the bottom of the previous stack frame.

Important

Code purists will no doubt notice that the prolog writes the saved registers area before the SP is adjusted to actually create space for the called function's stack frame. Normally, this is a dangerous thing. That's because an interrupt might fire, and its handler code will undoubtedly want to push and pop items from the stack, trashing the saved registers area. This potentially dangerous situation only occurs when building stack frames.

The solution is that the interrupt handler code immediately bumps the SP down by an amount that's larger than the saved registers area, and then does its work. This buffer zone value is currently 224 bytes, which is the amount of space required to store the processor's nonvolatile registers (nineteen 32-bit GPRs, and the eighteen 64-bit FPRs), rounded up to the nearest 8-byte boundary. If you're writing an interrupt handler and tempted to hard-code this value, don't. It is only valid for 32-bit implementations of the PowerPC processor, such as the 601, 602, 603, 603e, 604, and 604e. A 64-bit PowerPC processor, such as the PowerPC 620, requires a larger buffer zone because the save area must store the contents of 64-bit registers. Instead, use the Mac OS's Exception Handler to install the handler. The Exception Handler is privy to the processor type and adjusts the SP by the appropriate amount before calling the handler, and fixes up the SP after it exits.

When a function exits, special code called an epilog code tears down the stack frame. First, it increments the SP by the frame's size (thereby discarding it). Next it restores the registers from the called function's register save area. Then it restores LR from the calling function's link area. The RTOC is restored immediately by code in the calling function when execution returns to it.

To return from the called function, the epilog executes a branch instruction that uses the address stored in the LR register. This value is essentially a return address, and control passes to the calling function. (Now you know why these calling conventions bothered with that particular register!) As you can see, this arrangement requires that the called function doesn't mess with

the stack once the stack frame is built. However, the advantage to this scheme is that a function can both have local storage and access parameters without resorting to a frame pointer.

Background Info

Now that you've seen all the diagrams and read the explanation, how does this all translate into something you really appreciate, such as code? Here's some sample PowerPC assembly code, produced by the CodeWarrior compiler. This code is for a function that has no parameters, and doesn't return a result. The prolog to such a function is

```
void aFunction(void)

mflr  r0           // Get the link register (LR)
stw   r31, -4(SP)  // Store 1st non-volatile register in save area
stw   r30, -8(SP)  // Store 2nd non-voltaile register
stw   r0, 8(SP)    // Save LR in caller's link area (see Fig 4.8)
stwu  SP, -112(SP) // Allocate stack space and...
                   //    ...save previous SP in link area
```

The epilog for this the function is

```
lwz   r0,120(SP)   // Fetch LR from caller's
                   //     link area (stack frame size + 8)
addi  SP,SP,112    // Dispose of stack frame space
lwz   r31, -4(SP)  // Restore 1st non-volatile register
lwz   r30, -8(SP)  // Same for 2nd register
mtlr  r0           // Restore LR
blr                // Return to calling function via address in LR
```

Notice in the CodeWarrior prolog that the nonvolatile registers were saved first. Recall that the saved register area of the stack frame under construction is adjacent to the calling function's link area (or next to the base of its stack frame). Therefore, the saved registers area can be addressed by using simple negative offsets before the SP gets moved. These examples are based on code generated by the Metrowerks compiler. Other compilers can generate different prolog/epilog code.

Mode Mixing

If we continue with our example of an application calling a Toolbox routine, the real question becomes: Is the Toolbox routine about to be called implemented as 680x0 code or PowerPC code? Put another way, before the processor hops to that routine, how does it determine whether it should simply start fetching PowerPC instructions or call the 68LC040 emulator instead?

The solution is the Mixed Mode Manager. This is a set of routines that enables a PowerPC function to call a 680x0 function or a 680x0 function to call a PowerPC function. Basically, the Mixed Mode Manager operates as a stack transformation engine. Its job is to massage the stack so that the function parameters get passed to the target routine in the proper order. The problem is complicated by the fact that the calling conventions used by a 680x0 environment vary depending upon the programming language used (C, Pascal, and assembler each use a different method), whereas the PowerPC uses the register-based mechanism for all programming languages.

Apple solved this thorny problem by designing a Universal Procedure Pointer (UPP) for all exported functions. A 680x0 procedure pointer is normally the address of a function's entry point. A UPP has either the usual 680x0 procedure pointer (the routine's address) or the address of a routine descriptor. Take a glance at the "Types.h" header file:

```
#if     GENERATINGCFM
typedef struct RoutineDescriptor *UniversalProcPtr, **UniversalProcHandle;

#else
typedef ProcPtr UniversalProcPtr, *UniversalProcHandle;

#endif
```

The conditional statement `GENERATINGCFM` informs the CodeWarrior compiler to use a certain set of declarations when generating PowerPC code fragments. Otherwise, it uses the 680x0 declarations for code generation. For PowerPC code, the UPP declaration becomes a pointer to data structure. This data structure, `RoutineDescriptor`, contains information that enables the Mixed Mode Manager to make the context switch from one instruction set

architecture (ISA) to another. This structure is termed a *routine descriptor*, because it decribes the target routine's address, the number and size of the parameters passed to the routine, the language calling convention the routine uses, and what ISA implements the routine. Here's a closer look at `RoutineDescriptor`'s contents, as revealed by examining the header file "MixedMode.h":

```
// Routine Descriptor Structure
struct RoutineDescriptor {
        UInt16                  goMixedModeTrap;        // Trap word, 0xAAFE
        SInt8                           version;        // Current version
        RDFlagsType             routineDescriptorFlags; // Flags
        UInt32                  reserved1;              // Must be zero
        UInt8                           reserved2;      // Must be zero
        UInt8                           selectorInfo;
        UInt16                  routineCount;           // Number of routines
        RoutineRecord     routineRecords[1];            // Array of routines
};
typedef struct RoutineDescriptor RoutineDescriptor;

typedef RoutineDescriptor *RoutineDescriptorPtr, **RoutineDescriptorHandle;
```

Of course, this listing then begs the question of what a `RoutineRecord` is, and it too can be found in the "MixedMode.h" file:

```
// Mixed Mode Routine Records
typedef unsigned long ProcInfoType;

// ISA Types
typedef SInt8 ISAType;

enum {
        kM68kISA        = (ISAType)0,
        kPowerPCISA     = (ISAType)1
};

struct RoutineRecord {
        ProcInfoType            procInfo;
        UInt8                                   reserved1;      // Must be 0
        ISAType                         ISA;
        RoutineFlagsType routineFlags;                          // Flags
        ProcPtr                         procDescriptor;
        UInt32                          reserved2;              // Must be 0
```

```
        UInt32                          selector;
};
typedef struct RoutineRecord RoutineRecord;

typedef RoutineRecord *RoutineRecordPtr, **RoutineRecordHandle;
```

There are several key items to notice in `RoutineRecord`. `ProcInfoType` stores a description of the routine's calling conventions. `ISAType` indicates whether the routine exists as 680x0 or PowerPC code. Finally, `ProcPtr` points to the routine itself, unless it references a PowerPC routine, in which case it actually points to a transition vector.

When a 680x0 application calls a Toolbox routine, the following sequence of events occurs. First, execution passes through the UPP for the Toolbox call. This in turn goes to either the routine directly (if it's a 680x0 application calling a 680x0 routine) or to a routine descriptor. The head of the routine descriptor has a 680x0 trap word (that's right, a trap word) that invokes the Mixed Mode Manager. The Mixed Mode Manager uses the routine descriptor information to build a switch frame on the stack, just beneath the current stack frame. This switch frame contains the information necessary to transfer the arguments in the proper order to the target routine, plus the state of various registers in both the 680x0 and PowerPC environments (see Figures 4.9 and 4.10).

The routine descriptor then points to the routine's transition vector, which in turn points to the TOC and entry point of the routine in the Toolbox shared library. The Mixed Mode Manager uses the transition vector to adjust the RTOC and pass control to the Toolbox code (see Figure 4.11).

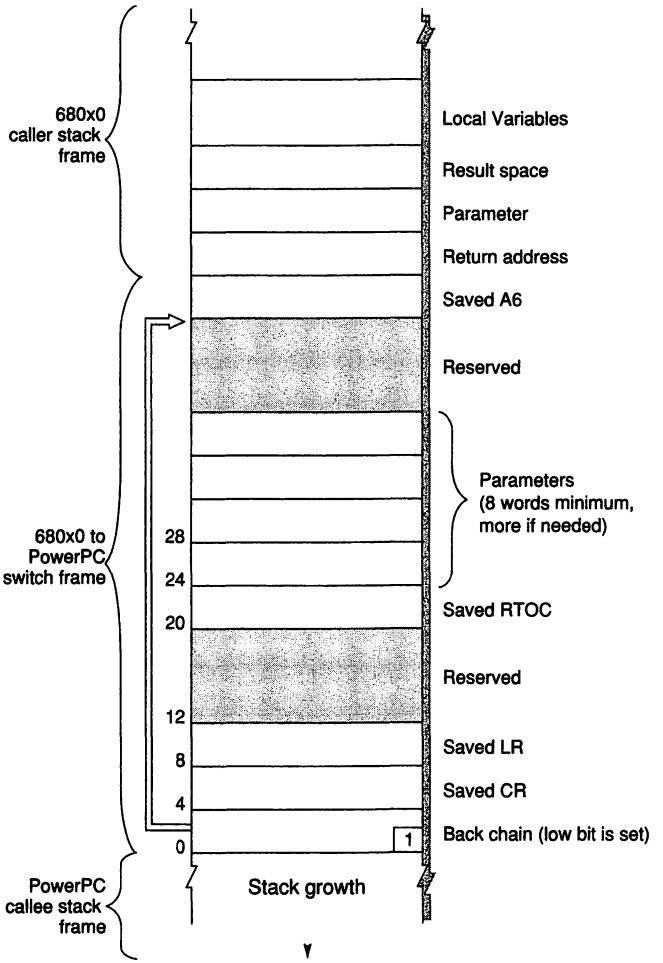


Figure 4.9 The PowerPC stack during a call from 680x0 application to a PowerPC routine.

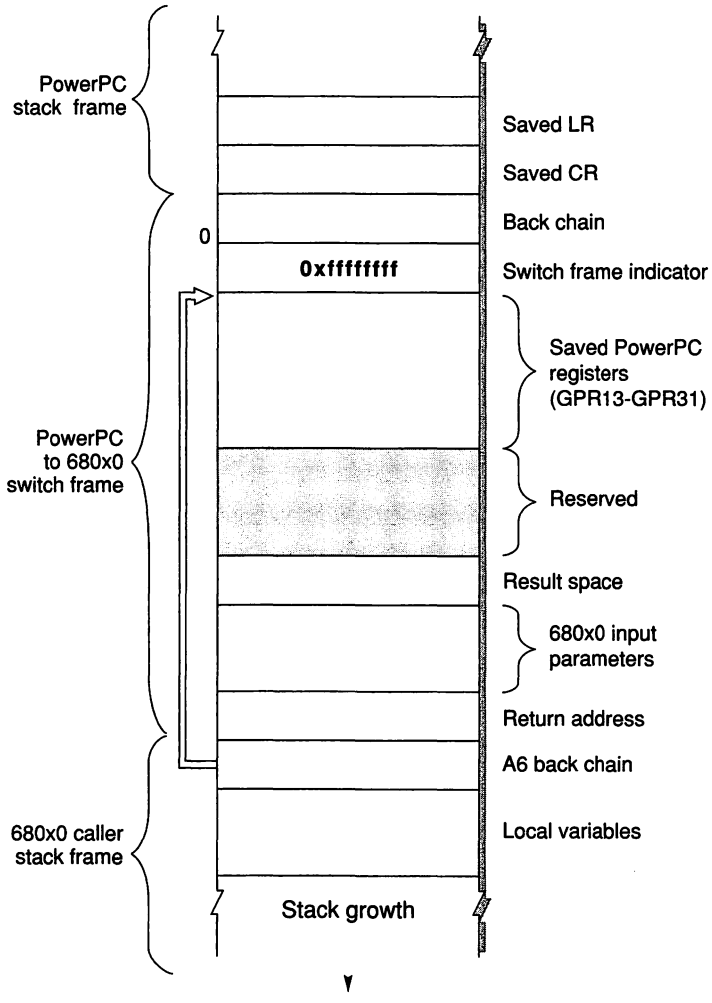


Figure 4.10 The PowerPC stack during a call from a native application to a 680x0 routine.

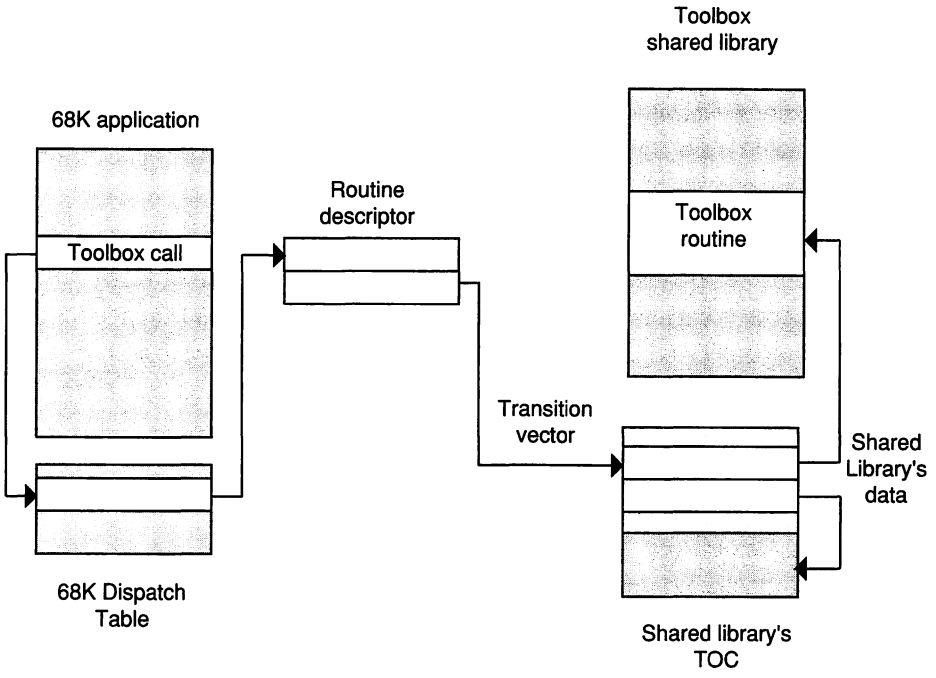


Figure 4.11 How a 680x0 application accesses a PowerPC Toolbox routine.

Important

Apple rewrote the most heavily used Toolbox calls in PowerPC code so that 680x0 applications could benefit from the native performance of the PowerPC processor. The Toolbox calls ported for the first Power Macs include portions of QuickDraw, the Font Manager, TrueType, QuickTime, the Resource Manager, the Memory Manager, fixed-point math, SANE, and the Script Manager (for foreign language support). This situation changes as Apple releases new versions of the Mac OS. For example, the System 7.5.2 release for the second-generation Power Macs, such as the Power Mac 9500, 8500, 7500, and 7200, contains even more native code. The heavily used Resource Manager is now completely native, as are the network protocol stacks, and some of the device drivers, especially the SCSI Manager. Other portions of the native Mac OS, such as QuickDraw, the math libraries, and the Memory Manager have been tuned for better performance. Additionally, the 68LC040 dynamic recompiling (DR) emulator uses code recompilation to make the remaining 680x0 code execute faster.

continues

continued

Please note that the revised SANE is available only to support 680x0 applications. Native PowerPC applications should use the new industry-standard C math libraries to access to PowerPC's floating-point hardware. See Appendix B for more information.

The Code Fragment Manager can use a code fragment's version information to allow updates for the Toolbox shared library located in ROM. Just as important, because some of the Macintosh OS is still 680x0-specific, the Trap Dispatcher and dispatch table is supported. This allows existing Extensions and Control Panels to patch the operating system as before.

Future Directions

As just mentioned, the operation of the native version of the Mac OS (currently 7.5.x) still relies on the 680x0 exception handler and trap words. Under Copland, this dispatch mechanism will completely change. The 680x0 processor-specific A-trap table scheme will be replaced completely by a processor-independent dispatch mechanism based on transition vectors. This confers some significant advantages. First, the granularity of what you patch can be finer. Under the old scheme, certain Toolbox calls use the same trap word as an entry point. Examples of such calls are `FSpOpenDF()` in the File Manager, which goes through the _HighLevelFSDispatch trap, or `FindFolder()` in the Alias Manager, which routes through _AliasDispatch. How these routines work is that a value—called a selector—gets passed to a base routine. The base routine uses this selector value to determine the actual routine that gets called. Patching such routines are messy because you first patch the base routine. The patch code then monitors the calls that the base routine receives to determine whether the routine being used is the one whose behavior you want to change. This arrangement can introduce all sorts of undesired side effects. Under Copland's new Patch Manager, you can actually patch only the `FSpOpenDF()` or `FindFolder()` routines. The second advantage is that the emulation overhead of the 680x0 Trap Dispatcher goes away, so that Toolbox calls are dispatched quickly. The catch is that all the existing Extension and Control Panel code based on the A-trap mechanism breaks.

The header files for the Toolbox calls contain UPPs for those routines written in PowerPC code. Thus, calls to these native routines bring in the Mixed Mode Manager to handle a context switch, when necessary. Because a UPP points either to a 680x0 routine address or to a transition vector, the same header files can be used by development tools on either 680x0- or PowerPC-based Macs. Normally, you won't be aware of the sleight of hand going on here, except in certain situations.

Hazard

If you're writing custom handlers that the operating system calls back to within your application (in Mac parlance these are termed "call-back functions"), you need to write your own UPP so that the Mixed Mode Manager can manage a context switch when that handler is called. The UPP is required because the operating system has no idea of what type of ISA your handler is written in, or the type of arguments it uses. Examples of such handlers include: custom controls for windows or dialogs that use a control definition function, event filters for dialogs or alerts, high-level event handlers, and plug-in modules. Otherwise, the Power Mac crashes and burns.

Fortunately, Apple provides special functions in the header files for those routines likely to need a UPP. These functions take the procedure pointer you pass to it and tack a routine descriptor on your custom function. For example, in Chapter 3, you saw that `NewAEEventHandlerProc()` helped install the high-level Apple Event functions. For custom event filter alerts and dialogs, there's `NewModalFilterProc()`, and so on. These functions help immensely in hiding the details of building a UPP from scratch. When writing a call-back function documented in *Inside Macintosh*, search the header files for the term "New" to locate these support functions. Why? As you can see in the two examples provided here, these functions always start with the term "New," so that's a good place to start to discover if there are ready-made functions available for your use.

Of course, making these context switches has a price. The Mixed Mode Manager has an overhead of 50-100 680x0 instructions when handling a context switch between the 680x0 and PowerPC environment. For certain heavily-called small Toolbox routines, this context switch overhead becomes

considerable, and can impact performance. For these routines, Apple actually implemented them as "fat traps." That is, these routines were written in both ISAs (680x0 and PowerPC code). This way, no matter what ISA calls the routine, it can be used without requiring the penalty of a context switch.

How is this done? The Mixed Mode Manager examines the UPP of the next routine to be called. If this routine uses the same ISA as the current routine as determined by looking at `ISAType` inside of `RoutineRecord`, the Mixed Mode Manager retains the current ISA environment. Notice that the routine descriptor contains an array of routine records. This allows a fat trap's routine descriptor to have two routine records: one that describes the calling conventions and the address of the 680x0-version of the routine, and one that points to the transition vector for the PowerPC-version of the routine.

The Mixed Mode Manager thus calls the appropriate code section in the routine. This in turn eliminates the overhead of the context switch. For example, if the processor is currently running in the 68L040 emulator, and this 680x0 code calls another Toolbox routine, the Mixed Mode Manager calls the corresponding 680x0 version of that routine (if it was written as a fat trap). As more of the Mac Toolbox is replaced by native PowerPC code, these Mixed Mode Manager context switches will become more infrequent and the applications will run faster.

A Tale of Two Processors

In this chapter, you learned about the Mac's application architecture for both 680x0-based and PowerPC-based Macs. While these architectures are quite different, the Power Macintosh's OS manages to support both. The PowerPC's run-time architecture provides a simplified structure that can run faster, as more applications and more of the Mac Toolbox get written as PowerPC code. It also has separated the data so that the operating system can become a full-blown preemptive multitasking OS in the future.

At the same time, the Power Mac OS can support existing 680x0 code and traps. This capability is provided by the use of special declarations in the Toolbox header files and in your code for custom functions. We'll see how this is done in the next chapter, "Putting It All Together."

Putting It All Together

Here's where you apply the knowledge gained in Chapter 4 to utilize parts of the Power Macintosh run-time architecture. The task might require rolling your own UPP for a custom function or calling the Code Fragment Manager itself.

In the last chapter, you saw that the Power Mac is quite different under the hood, from its PowerPC processor to the run-time architecture used by its native applications. However, by use of unique data structures such as routine descriptors and UPPs, plus special-purpose functions in the Toolbox header files, many of these differences are hidden from you.

Almost.

In this chapter you are going to explore situations that don't quite fall into a category that the header files can conveniently handle. You'll stray into this gray area while writing something exotic. Such exotic fare includes plug-in modules that expand the capabilities of an application, and extensions that enhance the operating system by adding patch code. Because the Power Mac's run-time architecture makes writing these types of objects easier, it's well worth knowing how to do this. These types of jobs require that you

have a firm grasp of the fundamentals that you learned in the last chapter. You'll see how this is accomplished with actual working code.

Let's take an example of writing a custom function first. Let's use a real case example here, where I wrote a custom function in a utility program called SwitchBank. I wrote SwitchBank out of my frustration with System 7.5 in dealing with "captive" CD-ROMs. A captive CD-ROM is where the Mac's File Sharing software mistakenly assumes that you're sharing it with other networked users. When you try to eject the CD-ROM, you get the message "The disk 'Your Favorite CD' could not be put away, because it's being shared," and the disk stays put. This is because the Mac OS tries to protect the networked users' access to the CD-ROM by refusing to eject it.

There are two ways File Sharing comes to this erroneous conclusion. First, in your eagerness to try out that new CD-ROM game, you insert the disk into the drive before the Macintosh completes booting. Or, the Mac crashes with the disk already in the drive. In either case, a feature of the Mac OS is that when it boots with File Sharing active and detects a CD-ROM in the drive, it assumes that you want to share its contents. Thus the Mac OS mounts the disk as a shared volume. This enables a Macintosh file server to resume sharing a CD-ROM such as the Oxford Dictionary after a power glitch. For you, however, the solution is to go to the Control Panels folder, open the File Sharing Setup Control Panel, and turn off File Sharing. Now you can eject the disk.

Later in the day, you're at the other end of the building. While talking with a coworker, you realize there's a file on your Mac you need to give her. Because of File Sharing, it's easy to use her Mac to log onto your Mac, and copy the file to her system, right? Wrong. To your dismay, you discover that you left File Sharing turned off, and so you have to walk back to your office anyway. Because I look at lots of beta software, this scenario happens more often that I care to admit. I finally decided to do something about it.

Important

This text was written using the full version of Metrowerks CodeWarrior. You'll have to use slightly different steps when using the limited version on the CD; the limited version can only work with the sample files provided on the CD, so the commands Add File... and New Project are not available.

So, if you are following along using the limited version of CodeWarrior that's on the CD, when the text tells you to use the New Project or the Add File... command, you should instead open the related project file and keep it open throughout the exercise. All the associated files will already be in the project so you won't need the Add File... command. Then, you can follow the same procedures as if you were using the full version of CodeWarrior.

SwitchBank: Initial Investigation and Design

Ideally, I wanted something that would switch off File Sharing long enough to eject the CD-ROM, and restart it. To control File Sharing, though, I first had to know something about it. Simply put, it's an Extension file that, when installed, makes each Mac look like a file server. This leads you to a question: What exactly does the File Sharing Extension do? The answer to that question is an interesting one. Even better, the answer was already available.

Remember the small program "process.c" from Chapter 2 that listed all of the running processes on the system? If you have File Sharing active, one of the processes it invariably lists is called the File Sharing Extension. This implies that an application actually implements File Sharing, because processes are running applications. To confirm this, I made a copy of the File Sharing Extension file, and opened it with ResEdit. There was the usual INIT resource, but sure enough, tucked in with the ICN#, BNDL, and other resources was a CODE resource. Opening the CODE resource, I saw a CODE resource 0. Could that be a jump table? When I examined that resource closely, I saw an array of numbers, where the value 0xA9F0 appeared frequently (see Figure 5.1). Because you know from Chapter 4 that this value is the `LoadSeg()` trap, it confirmed that this was indeed a 680x0 processor-specific jump table. The presence of a jump table in the Extension file meant that there was actually an application embedded in it. This was good news indeed, because you can easily control applications with high-level Apple Events.

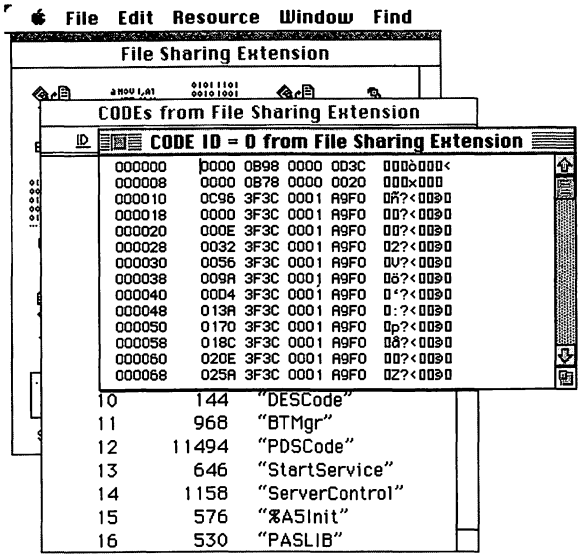


Figure 5.1 The CODE 0 jump table in the File Sharing Extension file.

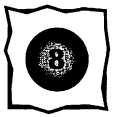
Background Info

Why isn't File Sharing written in native code? Remember, not all of the Macintosh Toolbox, which by a loose definition includes operating system software, got ported to PowerPC RISC code. This happens to include some of Apple's own extensions, including portions of QuickTime, the Apple CD-ROM driver, and others. This will change over time as Apple completes the porting process.

File Sharing is known as a *daemon* or, in Mac parlance, a faceless background application. The program has no user interface and, thus, no A5 world. But wait a minute, doesn't File Sharing use a Control Panel, which sports a user interface? The answer is that the Control Panels don't do much more than store the current settings and issue Apple Events to the daemons when you change these settings.

Such daemons turn out to be fairly common in the latest releases of the Mac OS. The Express Modem, PlainTalk voice recognition, and the LaserWriter 8.3 Desktop Printer Spooler all use daemons. If you're curious as to what OS

services use daemons, run the process program and save the list of active processes to a file. Now, turn on these services and use the process program again to see what processes appear in memory.



Future Directions

Under Copland, such daemons can be written as Copland-aware programs. Copland-awareness confers a number of advantages. First, the daemon runs in a separate memory space and is more robust because it is protected from wild accesses from malfunctioning applications that might cause memory corruption. The other advantage is that the OS kernel executes it through a preemptive time-slice scheduler, which means that the daemon still gets time to run, even when a processor-hogging application such as Munger is active. It's not surprising then, that Apple has implemented services such as voice recognition as daemons—it makes the move to Copland much easier.

SwitchBank's design is simple. It orders the File Sharing process to stop and ejects the CD-ROM. Once that's done, the program restarts the File Sharing application. Like the program itself, the user interface should be simple as well. The drag-and-drop feature you implemented in SonOMunger can be used here. You let the user drag the CD-ROM icon onto the SwitchBank icon to eject it. To encourage the program's use so that folks will readily drag the CD-ROM onto SwitchBank's icon, it should eject any volume dropped on it.

Building Resources with Rez

Use the same approach in building SwitchBank that you applied to Munger and SonOMunger. That is, you start by creating the interface resources first. However, you use a program called Rez to generate the resources this time around.

Rez is an MPW tool that accepts text statements that use a C-style syntax to describe a resource. It generates the appropriate binary image of each resource from these descriptions. While this method of resource building doesn't have the point-and-click flexibility of drawing your dialogs, alerts, and windows that ResEdit offers, it does have its advantages. For example, with an appropriately written Rez source file, you could modify the resource

ID numbers of all your dialog boxes and dialog items by editing a few definition statements and "recompiling" the file. That's a job that would require lots of pointing and clicking to fix in ResEdit. Also, large applications require sophisticated user interfaces, which in turn means complex resources. These sets of resources are easier to maintain as a Rez source file. Typically, you'll write most of your resources with Rez statements, and draw your icons in ResEdit. You then use the DeRez tool, which is a resource disassembler, to reduce the binary icon resources into text Rez statements.

To begin, launch the CodeWarrior IDE and open the file *SwitchBank.r* in the folder PPC Examples:SwitchBank. Or, you can type:

```
#include "SysTypes.r"
#include "Types.r"

#define AllItems  0b11111111111111111111111111111111
/* 31 flags */
#define NoItems       0b00000000000000000000000000000000
#define MenuItem1     0b00000000000000000000000000000001
#define MenuItem2     0b00000000000000000000000000000010
#define MenuItem3     0b00000000000000000000000000000100
#define MenuItem4     0b00000000000000000000000000001000

#define MENU_BAR_ID  128
/* Menu bar resource for your menus */
#define APPLE_MENU   128
/* Menu ID for Apple menu */
#define FILE_MENU    129
/* Menu ID for File menu */
#define EDIT_MENU    130
/* Menu ID for Edit menu */
#define SWITCH_MENU  131
/* Menu ID for File Share control */

#define ABOUT_BOX_ID 128
/* Resource IDs for your windows & dialogs */
#define ERROR_BOX_ID 130
#define ERROR_MESS_ID   131

#define APPL_FREF    128
/* Resource IDs for file refs & icons */
#define DISK_FREF    129
#define SWITCH_ICON  128
```

Notice the header files "SysTypes.r" and "Types.r." They supply declarations and structures that define the resource statements written here. Observe also that the definitions for the menu and dialog resource IDs are similar to those you used in SonOMunger. That shouldn't come as a surprise, because those definitions tell the program what resource, by its type and ID number, to use. You are using those exact numbers here to generate similar resources. In fact, some programmers take the definitions in this section and move them into a separate header file that both the program code and Rez source draw on for resource information. The other reason the definitions appear the same is that you are going to reuse a lot of SonOMunger's code.

Now it's time to write some resource descriptions. Locate the following text in SwitchBank.r, or type:

```
/* Version info for the Finder's Get Info box
resource 'vers' (1, purgeable)
{
   0x01,
   0x10,
   beta,
   0x00,
   verUs,
   "1.1ß",
   "1.1ß, by Tom Thompson"
};

/* Menu resources */
resource 'MBAR' (MENU_BAR_ID, preload)
{
  { APPLE_MENU, FILE_MENU, EDIT_MENU, SWITCH_MENU };
};

resource 'MENU' (APPLE_MENU, preload)
{
   APPLE_MENU, textMenuProc,
   AllItems & ~MenuItem2,
/* Disable separator line, enable About Box */
   enabled, apple,
   {
      "About SwitchBank 1.1…", noicon, nokey, nomark, plain;
      "-",                noicon, nokey, nomark, plain
   }
```

```
};

resource 'MENU' (FILE_MENU, preload)
{
   FILE_MENU, textMenuProc,
   AllItems,
   enabled, "File",
   {
      "Quit",           noicon, "Q", nomark, plain
   }
};

resource 'MENU' (EDIT_MENU, preload)
{
   EDIT_MENU, textMenuProc,
   AllItems & ~MenuItem2,
/* Disable separator line */
   enabled, "Edit",
   {
      "Undo",           noicon, "Z", nomark, plain;
      "-",              noicon, nokey, nomark, plain;
      "Cut",            noicon, "X", nomark, plain;
      "Copy",           noicon, "C", nomark, plain;
      "Paste",          noicon, "V", nomark, plain
   }
};

resource 'MENU' (SWITCH_MENU, preload)
{
   SWITCH_MENU, textMenuProc,
   AllItems,
   enabled, "Controls",
   {
     "Toggle File Sharing", noicon, "T", nomark, plain
   }
};
```

To get the ß symbol in the 'vers' resource, press Option-S.

The previous statements describe your menu resources. They define a resource type ('MENU' and 'MBAR'), its ID number, and certain attributes. They also describe the menu's title, and its item list. The item list contains the text of each menu item, and a description of how it appears in the menu.

For instance, the Controls menu has a single item called Toggle File Sharing that's displayed with no accompanying icon, no checkmark, and in plain text. It has a Command key equivalent that is the character "T." The 'vers' resource provides the version number information that appears in a file's Info box.

```
/* Your error messages */
resource 'STR#'(128, purgeable)
{
   {
   /* [1] */ "A problem occurred stopping File Sharing.";
   /* [2] */ "A problem occurred starting File Sharing.";
   /* [3] */ "A problem occurred while ejecting the volume.";
   /* [4] */ "You can't eject the startup volume.";
   /* [5] */ "Couldn't find the startup volume.";
   /* [6] */ "Couldn't get valid system information.";
   /* [7] */ "Couldn't locate the File Sharing Extension file.";
   /* [8] */ "A problem occurred while loading the Apple Event */
          */ handlers.";
   /* [9] */   "Sorry, SwitchBank requires System 7 or later */
          */ to run.";
   }
};
```

These are your error messages stored as Pascal strings in a STR# resource. You place them here, rather than hard-coding them as you did in SonOMunger, for a good reason. As a list in a resource, these strings can be easily modified with ResEdit without having to recompile the program code. This opens the possibility of your program being translated into foreign languages. You can have someone use ResEdit to edit the menu lists, dialog boxes, and error messages so that they appear in another language (say, French) without changing the executable code.

```
/* This ALRT and DITL are used as an About Box */
resource 'DLOG' (ABOUT_BOX_ID, purgeable)
{
   {31, 6, 224, 265},
   altDBoxProc,
   visible,
   noGoAway,
   0x0,                /* No refCon */
   ABOUT_BOX_ID,
```

```
   ""                    /* No window title */
};

resource 'DITL' (ABOUT_BOX_ID, purgeable)
{
   {
      /* Item 1 */
      {154, 80, 175, 180},
      Button { enabled, "OK" },
      /* Item 2 */
      {4, 68, 38, 193},
      StaticText { disabled, " SwitchBank 1.1\nby Tom Thompson" },
      /* Item 3 */
      {86, 11, 102, 250},
      StaticText { disabled, " Copyright © 1994 Tom Thompson." },
      /* Item 4 */
      {44, 114, 76, 146},
      Icon { disabled, SWITCH_ICON },
      /* Item 5 */
      {107, 43, 133, 217},
      StaticText { disabled, "Written in Metrowerks C " }
   }
};

/* The ALRT and DITL for the basic error screen */
resource 'ALRT' (ERROR_BOX_ID, purgeable)
{
   {40, 40, 127, 273},
   ERROR_BOX_ID,
   {
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent
   }
};

resource 'DITL' (ERROR_BOX_ID, purgeable)
{
   {
      { 52, 162, 72, 220 },
      Button { enabled, "OK" },
      { 54, 17, 70, 151 },
```

```
      StaticText { disabled, "I/O error, ID = ^0" }
   }
};

/* Alert and DITL for error message screen */
resource 'ALRT' (ERROR_MESS_ID, purgeable)
{
   { 40, 40, 147, 280 },
   ERROR_MESS_ID,
   {
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent
   }
};

resource 'DITL' (ERROR_MESS_ID, purgeable)
{
   {
      { 73, 168, 93, 226 },   Button { enabled, "OK" },
      { 53, 14, 97, 157 },    StaticText { disabled, "^0" }
   }
};

/* File reference resources */
resource 'FREF' (DISK_FREF)
{
   'disk',
   1,
   ""
};


resource 'FREF' (APPL_FREF)
{
   'APPL',
   0,
   ""
};

/* Bundle resource */
resource 'BNDL' (128)
```

```
{
   'SWCH', 0,
   {
   'ICN#', { 0, SWITCH_ICON },
/* Only 1 icon */
   'FREF', { 0, APPL_FREF, 1, DISK_FREF }
/* Two types of files */
   }
};

/* Signature resource - all 'STR ' resources */
/* must be declared before this! */
type 'SWCH' as 'STR ';

resource 'SWCH' (0) {
   "SwitchBank 1.1ß"
};
```

These statements describe your alerts, dialog boxes, and their dialog item lists. There's also the bundle resource, BNDL, and its satellite definitions in the FREF resources that describe the application's file type, and a disk type. This latter type allows file type filtering similar to what's used for SonOMunger. That is, you can only drag and drop icons representing TEXT file types onto the SonOMunger's icon, and for SwitchBank you can only drag and drop an icon representing a disk (or volume) onto its icon. This filtering action performed by System 7 is very convenient. An application won't see a high-level Open Document Apple Event unless the Mac OS deems that the dropped file type matches what the application can handle.

```
/* Your icon data */
data 'ICON' (SWITCH_ICON)
{
   $"7FFF FFFE 4000 0002 5C00 003A 55F8 1FAA"
   $"5D08 10BA 4108 1082 4108 1082 4108 1082"
   $"41B8 1D82 4110 0882 4110 0882 4110 0882"
   $"471C 38E2 4514 28A2 4514 28A2 4514 28A2"
   $"471C 38E2 4110 0882 411F F882 4110 0882"
   $"4110 0882 4110 0882 41FF FF82 4004 2002"
   $"4004 2002 4004 2002 4004 2002 5C04 203A"
   $"5404 202A 5C07 E03A 4000 0002 7FFF FFFE"
};
```

```
data 'ICN#' (SWITCH_ICON)
{
   $"7FFF FFFE 4000 0002 5C00 003A 55F8 1FAA"
   $"5D08 10BA 4108 1082 4108 1082 4108 1082"
   $"41B8 1D82 4110 0882 4110 0882 4110 0882"
   $"471C 38E2 4514 28A2 4514 28A2 4514 28A2"
   $"471C 38E2 4110 0882 411F F882 4110 0882"
   $"4110 0882 4110 0882 41FF FF82 4004 2002"
   $"4004 2002 4004 2002 4004 2002 5C04 203A"
   $"5404 202A 5C07 E03A 4000 0002 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
};

/* SwitchBank's color icon in icl8 format */
data 'icl8' (SWITCH_ICON)
{
   $"00FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FF00"
   $"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A"
   $"2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
   $"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2A2A 2A2A"
   $"2A2A 2A2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
   $"00FF 2AFF 2AFF 2AFF FFFF FFFF FF2A 2A2A"
   $"2A2A 2AFF FFFF FFFF FF2A FF2A FF2A FF00"
   $"00FF 2AFF FFFF 2AFF F52A F52A FF2A 2A2A"
   $"2A2A 2AFF F52A F52A FF2A FFFF FF2A FF00"
   $"00FF 2A2A 2A2A 2AFF 2A2A 2A2A FF2A 2A2A"
   $"2A2A 2AFF 2A2A 2A2A FF2A 2A2A 2A2A FF00"
   $"00FF 2A2A 2A2A 2AFF 5454 5454 FF2A 2A2A"
   $"2A2A 2AFF 5454 5454 FF2A 2A2A 2A2A FF00"
   $"00FF 2A2A 2A2A 2AFF 7F7F 7F7F FF2A 2A2A"
   $"2A2A 2AFF 7F7F 7F7F FF2A 2A2A 2A2A FF00"
   $"00FF 2A2A 2A2A 2AFF FF7F FFFF FF2A 2A2A"
   $"2A2A 2AFF FFFF 7FFF FF2A 2A2A 2A2A FF00"
```

```
$"00FF 2A2A 2A2A 2AFF 7F7F 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF7F 7F7F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 5454 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF54 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2AFF FFFF 2A54 7FFF FFFF 2A2A"
$"2A2A FFFF FF2A 547F FFFF FF2A 2A2A FF00"
$"00FF 2A2A 2AFF F5FF 2A54 7FFF F5FF 2A2A"
$"2A2A FFF5 FF2A 547F FFF5 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF 54FF 2A54 7FFF 54FF 2A2A"
$"2A2A FF54 FF2A 547F FF54 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF 54FF 2A54 7FFF 54FF 2A2A"
$"2A2A FF54 FF2A 547F FF54 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF FFFF 2A54 7FFF FFFF 2A2A"
$"2A2A FFFF FF2A 547F FFFF FF2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF FFFF FFFF"
$"FFFF FFFF FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF F52A F52A"
$"F52A F52A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 5454 5454"
$"5454 5454 FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 7F7F 7F7F"
$"7F7F 7F7F FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 54F5"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 542A"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 54F5"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 542A"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
$"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2AFF 54F5"
$"2A7F FF2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
$"00FF 2AFF 2AFF 2A2A 2A2A 2A2A 2AFF 542A"
$"2A7F FF2A 2A2A 2A2A 2A2A FF2A FF2A FF00"
$"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2AFF FFFF"
$"FFFF FF2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A"
```

```
    $"2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
    $"00FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FF00"
};

/* SwitchBank's color icon, in cicn format */
data 'cicn' (SWITCH_ICON)
{
    $"0000 0000 8010 0000 0000 0020 0020 0000"
    $"0000 0000 0000 0048 0000 0048 0000 0000"
    $"0004 0001 0004 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0004 0000 0000 0020 0020"
    $"0000 0000 0004 0000 0000 0020 0020 0000"
    $"0000 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 4000 0002 5C00 003A 55F8"
    $"1FAA 5D08 10BA 4108 1082 4108 1082 4108"
    $"1082 41B8 1D82 4110 0882 4110 0882 4110"
    $"0882 471C 38E2 4514 28A2 4514 28A2 4514"
    $"28A2 471C 38E2 4110 0882 411F F882 4110"
    $"0882 4110 0882 4110 0882 41FF FF82 4004"
    $"2002 4004 2002 4004 2002 4004 2002 5C04"
    $"203A 5404 202A 5C07 E03A 4000 0002 7FFF"
    $"FFFE 0000 0000 0000 0005 0000 FFFF FFFF"
    $"FFFF 0001 CCCC CCCC FFFF 0002 9999 9999"
    $"FFFF 0003 6666 6666 CCCC 0004 EEEE EEEE"
    $"EEEE 000F 0000 0000 0000 0FFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFF0 0F11 1111 1111"
    $"1111 1111 1111 1111 11F0 0F1F FF11 1111"
    $"1111 1111 1111 11FF F1F0 0F1F 1F1F FFFF"
    $"F111 111F FFFF F1F1 F1F0 0F1F FF1F 4141"
    $"F111 111F 4141 F1FF F1F0 0F11 111F 1111"
    $"F111 111F 1111 F111 11F0 0F11 111F 2222"
    $"F111 111F 2222 F111 11F0 0F11 111F 3333"
    $"F111 111F 3333 F111 11F0 0F11 111F F3FF"
    $"F111 111F FF3F F111 11F0 0F11 111F 333F"
    $"1111 1111 F333 F111 11F0 0F11 111F 223F"
```

```
   $"1111 1111 F223 F111 11F0 0F11 111F 123F"
   $"1111 1111 F123 F111 11F0 0F11 1FFF 123F"
   $"FF11 11FF F123 FFF1 11F0 0F11 1F4F 123F"
   $"4F11 11F4 F123 F4F1 11F0 0F11 1F2F 123F"
   $"2F11 11F2 F123 F2F1 11F0 0F11 1F2F 123F"
   $"2F11 11F2 F123 F2F1 11F0 0F11 1FFF 123F"
   $"FF11 11FF F123 FFF1 11F0 0F11 111F 123F"
   $"1111 1111 F123 F111 11F0 0F11 111F 123F"
   $"FFFF FFFF F123 F111 11F0 0F11 111F 123F"
   $"4141 4141 F123 F111 11F0 0F11 111F 123F"
   $"2222 2222 F123 F111 11F0 0F11 111F 123F"
   $"3333 3333 F123 F111 11F0 0F11 111F FFFF"
   $"FFFF FFFF FFFF F111 11F0 0F11 1111 1111"
   $"1F24 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F21 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F24 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F21 13F1 1111 1111 11F0 0F1F FF11 1111"
   $"1F24 13F1 1111 11FF F1F0 0F1F 1F11 1111"
   $"1F21 13F1 1111 11F1 F1F0 0F1F FF11 1111"
   $"1FFF FFF1 1111 11FF F1F0 0F11 1111 1111"
   $"1111 1111 1111 1111 11F0 0FFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFF0"
};

/* The system's color caution alert icon */
data 'cicn' (2)
{
   $"0000 0000 8010 0000 0000 0020 0020 0000"
   $"0000 0000 0000 0048 0000 0048 0000 0000"
   $"0004 0001 0004 0000 0000 0000 0000 0000"
   $"0000 0000 0000 0004 0000 0000 0020 0020"
   $"0000 0000 0004 0000 0000 0020 0020 0000"
   $"0000 0001 8000 0003 C000 0007 E000 0007"
   $"E000 000F F000 000F F000 001F F800 001F"
   $"F800 003F FC00 003F FC00 007F FE00 007F"
   $"FE00 00FF FF00 00FF FF00 01FF FF80 01FF"
   $"FF80 03FF FFC0 03FF FFC0 07FF FFE0 07FF"
   $"FFE0 0FFF FFF0 0FFF FFF0 1FFF FFF8 1FFF"
   $"FFF8 3FFF FFFC 3FFF FFFC 7FFF FFFE 7FFF"
   $"FFFE FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF 0001 8000 0003 C000 0003 C000 0006"
   $"6000 0006 6000 000C 3000 000C 3000 0018"
   $"1800 0019 9800 0033 CC00 0033 CC00 0063"
```

```
    $"C600 0063 C600 00C3 C300 00C3 C300 0183"
    $"C180 0183 C180 0303 C0C0 0303 C0C0 0603"
    $"C060 0601 8060 0C01 8030 0C00 0030 1800"
    $"0018 1801 8018 3003 C00C 3003 C00C 6001"
    $"8006 6000 0006 C000 0003 FFFF FFFF 7FFF"
    $"FFFE 0000 0000 0000 0006 0000 FFFF FFFF"
    $"FFFF 0001 FFFF CCCC 3333 0002 CCCC 9999"
    $"0000 0003 9999 6666 0000 0004 3333 3333"
    $"3333 0005 BBBB BBBB BBBB 000F 0000 0000"
    $"0000 0000 0000 0000 000F F000 0000 0000"
    $"0000 0000 0000 0000 004F F400 0000 0000"
    $"0000 0000 0000 0000 05FF FF50 0000 0000"
    $"0000 0000 0000 0000 04F3 3F40 0000 0000"
    $"0000 0000 0000 0000 5FF1 1FF5 0000 0000"
    $"0000 0000 0000 0000 4F31 13F4 0000 0000"
    $"0000 0000 0000 0005 FF11 11FF 5000 0000"
    $"0000 0000 0000 0004 F311 113F 4000 0000"
    $"0000 0000 0000 005F F12F F21F F500 0000"
    $"0000 0000 0000 004F 314F F413 F400 0000"
    $"0000 0000 0000 05FF 11FF FF11 FF50 0000"
    $"0000 0000 0000 04F3 11FF FF11 3F40 0000"
    $"0000 0000 0000 5FF1 11FF FF11 1FF5 0000"
    $"0000 0000 0000 4F31 11FF FF11 13F4 0000"
    $"0000 0000 0005 FF11 11FF FF11 11FF 5000"
    $"0000 0000 0004 F311 11FF FF11 113F 4000"
    $"0000 0000 005F F111 11FF FF11 111F F500"
    $"0000 0000 004F 3111 11FF FF11 1113 F400"
    $"0000 0000 05FF 1111 11FF FF11 1111 FF50"
    $"0000 0000 04F3 1111 114F F411 1111 3F40"
    $"0000 0000 5FF1 1111 112F F211 1111 1FF5"
    $"0000 0000 4F31 1111 111F F111 1111 13F4"
    $"0000 0005 FF11 1111 1112 2111 1111 11FF"
    $"5000 0004 F311 1111 1111 1111 1111 113F"
    $"4000 005F F111 1111 112F F211 1111 111F"
    $"F500 004F 3111 1111 11FF FF11 1111 1113"
    $"F400 05FF 1111 1111 11FF FF11 1111 1111"
    $"FF50 04F3 1111 1111 112F F211 1111 1111"
    $"3F40 5FF1 1111 1111 1111 1111 1111 1111"
    $"1FF5 FF31 1111 1111 1111 1111 1111 1111"
    $"13FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF 5FFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFF5"
};
```

Well, you probably won't type all of the hexadecimal codes here that define the color data of SwitchBank's icons, but you get the idea. The ICON resource defines a black-and-white icon, 32 pixels to a side. ICON is the great-granddaddy of the icon formats, starting on the original Mac in 1984. The cicn resource is a color icon format first introduced on the Mac II in 1987. It defines both a black-and-white icon, and an 8-bit color icon. It's not commonly used these days, because its complex format impairs fast data access. You supply it here because the Dialog Manager has a special feature that it uses when a dialog item is an icon. If the icon's cicn resource is available, the Dialog Manager substitutes the color icon for the dialog box's item icon, instead of using the black-and-white one. No special programming is required for this to occur. Your About Box uses an icon and it appears in color when you provide this cicn resource. This is also why you supply the cicn for the system's caution alert icon: When an alert appears, the icon appears in color on a color Mac.

The more prevalent color icon format is the icl8 format, which represents a large (32 pixels per side) 8-bit color icon. There's also a small (16 pixels per side) 8-bit color icon format called ics8 that's used to display file icons in the Apple menu, the Application menu, and the Standard File Dialog box. The Finder also uses the ics8 format if you set the window to display small icons. These formats define color data only, so access to the icon data is fast. If you've used ResEdit to spelunk around in other application resources, you can see that the Mac OS uses the icl8 and ics8 format to display file icons. For simplicity, I've omitted the ics8 icon data.

Save this editor window as the file "SwitchBank.r," or copy the file from the CD-ROM from CodeWarrior:Code Examples PPC:SwitchBank: folder. Like the convention of ending project file names with a .µ or .prj extension, Rez source files typically end with a .r extension. Now it's time to compile the Rez source code into resources.

Using Rez with the ToolServer

To create the resources, you'll need to use the Rez tool. You can do this in one of two ways: through the ToolServer application, or using CodeWarrior's plug-in version of Rez. You start first with the ToolServer application. A brief explanation is in order here. Apple's MPW software uses an application called the MPW Shell, which serves as an IDE for Apple's development tools. Where MPW differs from Metrowerks CodeWarrior is that many of its development

operations—such as compiling, linking, and building resources— are controlled by command lines typed into a Worksheet window. This window is managed by the MPW Shell. MPW tools, like Rez, are applications that have specialized or little interface code, and thus rely upon the environment set up by the MPW Shell to function. ToolServer is a special-purpose application that mimics this environment adequately so that these tools can operate outside of the MPW Shell. This makes them available to third-party vendors, which in turn lets CodeWarrior programmers tap into the large suite of MPW tools written over the years.

The first step in generating the resource, then, is to start the ToolServer. Go to the Tools menu in CodeWarrior's IDE and select Start ToolServer. A ToolServer Worksheet window appears, as does a new icon in the menu bar that represents the ToolServer's menu, and a Tool status window (see Figure 5.2).

The status pane within the Worksheet's bottom scroll bar shows what tool or script is active. If you're familiar with the MPW enviroment, you can type in the name of a tool and any arguments, then press Enter to start it. (Note: You must press the Enter key on the Extended keyboard, or Command-Return on the Standard keyboard. Pressing the Return key won't have any effect.) While you won't be working with ToolServer this way, you will rely on the output that appears in the Worksheet window to tell you if something's gone wrong. From the ToolServer menu, select ToolServer Tools, and a hierarchal menu appears, as shown in Figure 5.3.

Choose Rez from this menu. The Rez options window appears, as shown in Figure 5.4.

Important

Both the MPW Shell and ToolServer use low-level operations to start your test application or run an MPW Tool, such as Rez. Debugging programs such as Jasik Design's The Debugger can seize control when they mistake this low-level activity as a program exception. To prevent The Debugger from paying you a visit when you launch MPW tools from the ToolServer Tools menu, pick the tool `InformDbgr` from this menu first. It adjusts the run-time environment so that The Debugger doesn't interrupt your use of ToolServer. Or, you can edit the "MWStartup" file in the ToolServer folder to call the `InformDbgr` tool automatically when you start ToolServer.

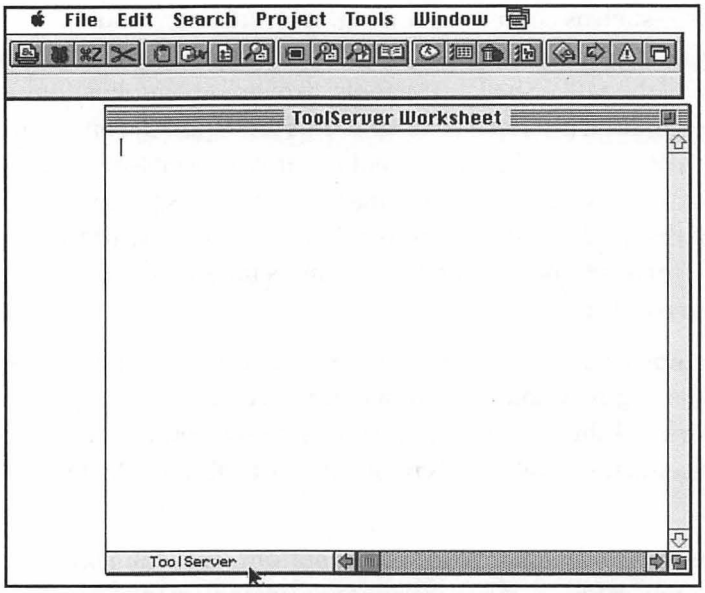


Figure 5.2 The ToolServer Worksheet window; the status pane in the lower left scroll bar shows the active tool.

Notice that the Type item is highlighted in this dialog box. This and the next item, Creator, are used to specify the type and creator of the file that Rez generates. Type **rsrc** in the Type item, press Tab to select the Creator item, and type **RSED**. RSED is ResEdit's creator signature, so once the output file is made, you can double-click on it to launch ResEdit and examine it immediately.

Now go to the popup menu labeled with the default file name of Rez.out, and select Write Output to a New File from this menu. A Standard File dialog box appears. First ensure that you're in the SwitchBank folder. Type the name **SwitchBank.µ.rsrc** and press Return. Now that you've selected the output file's name, type, and creator, let's specify the input. Start by clicking on the Files & Paths button. Rez places another window titled Files & Paths... on the screen (see Figure 5.5). You next guide Rez to the directory that contains the header files "SysTypes.r" and "Types.r." To do this, you click on the #Include Paths... button.

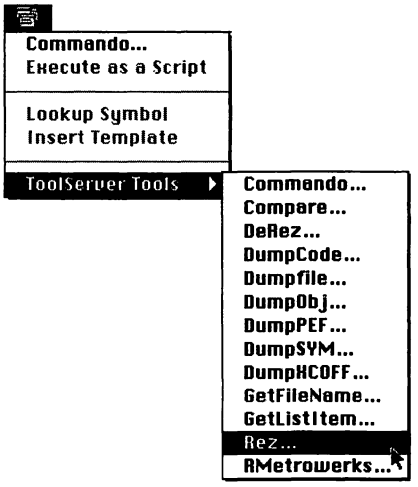


Figure 5.3 Some MPW tools available from the ToolServer.

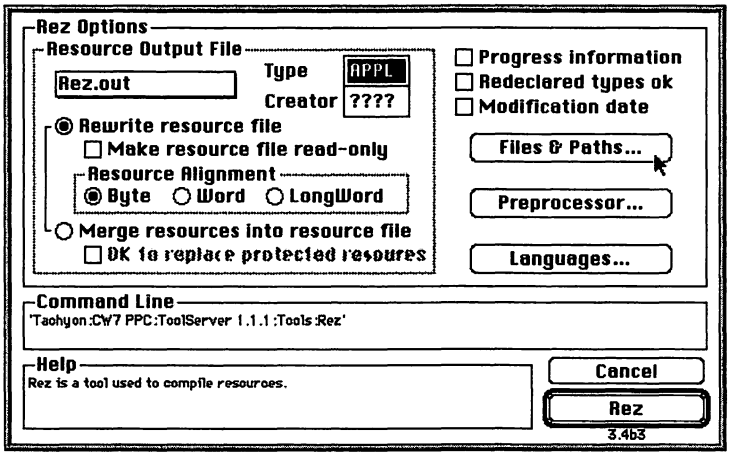


Figure 5.4 The Rez Options window, where the output file's name, type, and creator are set.

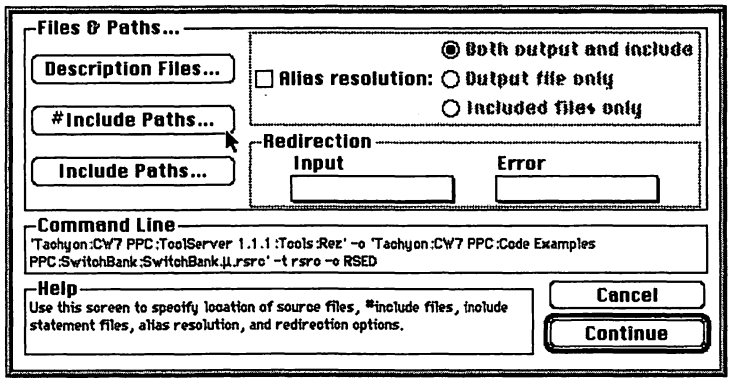


Figure 5.5 The Files & Paths window, where the input file and search paths are selected.

Locate the folder RIncludes in the CodeWarrior:ToolServer:Interfaces path. (The ToolServer folder and its contents are either on the CD-ROM or on your Mac's hard drive, depending upon the type of software installation you did.) When you get there, click on the Add Current Directory button, and this name gets appended into a list at the bottom of the window (see Figure 5.6).

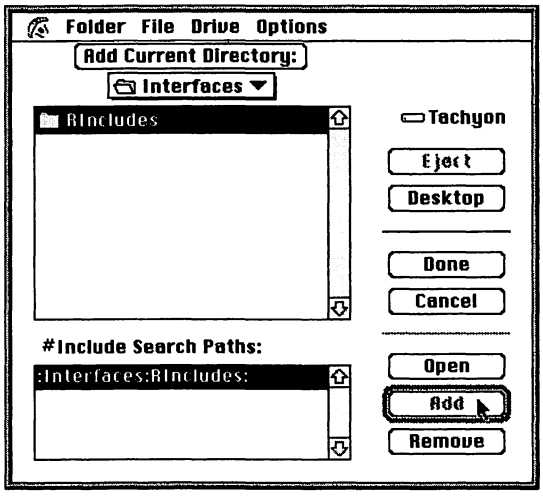


Figure 5.6 Adding an include file directory to the file search path.

Click the Done button, and you're returned to the Files & Paths window. Now go to the Redirection item in this window and click on the Input popup menu. Select Existing File from this menu, and a Standard File dialog box appears. Choose the file SwitchBank.r, and press Return. Finally, click on the Continue button to exit the Files & Paths window. Check that everything is set properly, then click on the Rez button. You should hear some hard disk activity, and the status pane indicates that Rez is active. No news in the Worksheet window is good news: it means the resource compilation ran successfully. Once the cursor changes from the rotating "beach ball" busy indicator back to an arrow, click on the WorkSheet window. You wind up back in CodeWarrior, but the ToolServer window remains open in case you want to run other MPW tools. If you're finished with Rez, go to the Tools menu in Metrowerks CodeWarrior and pick Stop ToolServer. The Worksheet window and the menubar icon disappear when the ToolServer application quits. You do this because ToolServer uses up to 1.5 MB of RAM, which you'll want to put to use elsewhere.

Background Info

If the ToolServer complains of missing files or scripts, or the MPW tools don't appear in the hierarchal menu, the ToolServer software may have been improperly installed. Both the CodeWarrior IDE and ToolServer require certain files be inside of specific directories to operate properly. Check the CodeWarrior documentation file, "CW & ToolServer," to determine whether this is the problem.

A file named "SwitchBank.µ.rsrc" should be present in the SwitchBank folder. Double-click it to launch ResEdit, and examine the resources. If everything appears in order, then it's time to start writing code.

Using Rez from Inside CodeWarrior

With CodeWarrior 1.4, Rez has become a plug-in module like the PPC C/C++ compiler, the PPC Linker, the MW Pascal PPC compiler, and a medley of other tools. In this case, adding a resource text file is just like adding a source

code file. And like a source code file, the CodeWarrior IDE manages its bookkeeping. If you edit the resource text file, CodeWarrior automatically compiles the resource text with Rez when you bring the project up to date. To add SwitchBank.r to the project, simply choose Add Files from the Project menu, and follow the same procedure that you did to add .c or .rsrc files to the project. Your project window should appear as in Figure 5.7.

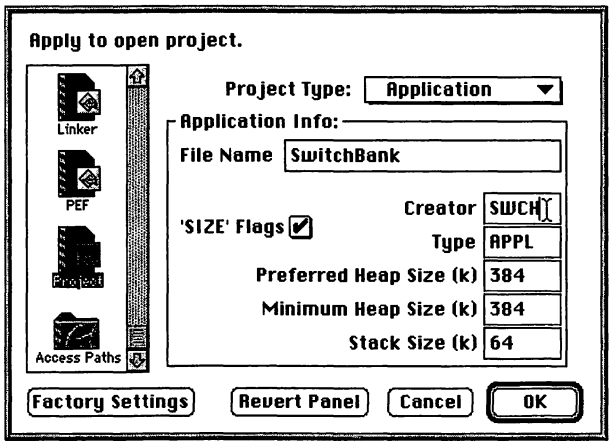


Figure 5.7 The SwitchBank project window with a resource text file added.

When you build the project or bring it up to date, you'll see the message "Compiling SwitchBank.r" in the status line of the CodeWarrior Toolbar. This is followed by the .r interface file name, just as the header files do when the C compiler operates on your source code. However, you won't see a separate SwitchBank.µ.rsrc file in the folder as you did with Rez under the ToolServer. Instead, the resources are added to the project file itself.

If the .r file doesn't appear in the Standard File dialog box, the Rez plug-in might not have been installed correctly. Go to the Edit menu, select Preferences, and check that there is a Rez panel present, as shown in Figure 5.8.

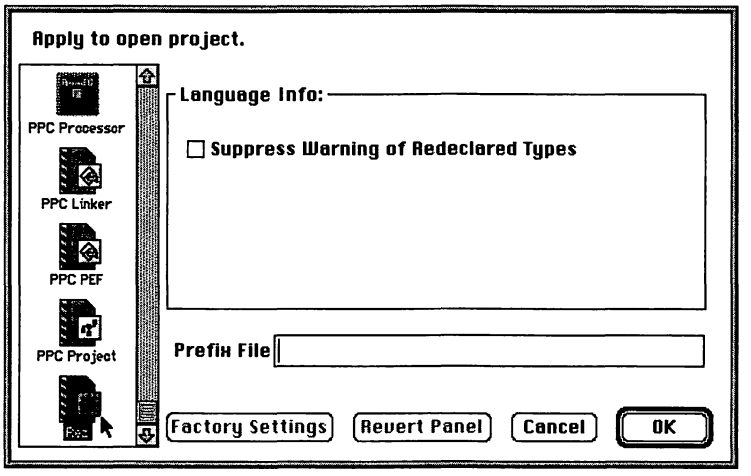


Figure 5.8 The Rez preferences panel in the CodeWarrior IDE.

If Rez isn't present, check the contents of the CodeWarrior Plug-ins folder to see whether the Rez plug-in is installed. The CodeWarrior IDE expects to find plug-in files in specific locations, and if they are absent, the services of those plug-ins aren't available to you. Consult the CodeWarrior documentation for the proper installation of the Rez plug-in.

Which method is better, using Rez with the ToolServer, or using the Rez plug-in? There is no best way, because both produce the desired result of creating resources for the program. If you want to double-check the results of your Rez compile, going the ToolServer route produces a file that you can examine later with ResEdit. The Rez plug-in stuffs the resources into the project file's data fork, which are not easily accessible. The Rez plug-in, however, is a lot easier to use, and, because the resource text file is part of the project, you can use the CodeWarrior editor to modify the file. Furthermore, the CodeWarrior IDE performs version tracking, just as it does with the C source and header files, and automatically calls Rez to update the resources after you make a change. Which method you choose depends entirely upon issues such as the migration from an MPW environment, or if another team member designs the resources separately from the coding efforts.

The SwitchBank Program

Let's start with the definitions first:

```
#include <Types.h>
#include <ConditionalMacros.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Controls.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Devices.h>
#include <Memory.h>
#include <Files.h>
#include <Events.h>
#include <Desk.h>
#include <OSEvents.h>
#include <ToolUtils.h>
#include <DiskInit.h>
#include <Folders.h>

#include <AppleTalk.h>
#include <AppleEvents.h>
#include <EPPC.h>
#include <PPCToolBox.h>
#include <Processes.h>

/* Definitions */
#define LAST_MENU      4
/* Number of menus */
#define LAST_HANDLER   3
/* Number of Apple Event handlers - 1 */

#define MENU_BAR_ID    128
/* ID for MBAR resource */
#define APPLE_MENU     128
/* Menu ID for Apple menu */
#define FILE_MENU      129
/* Menu ID for File menu */
#define EDIT_MENU      130
/* Menu ID for Edit menu */
#define SWITCH_MENU    131
```

```
/* Menu ID for File Share control */
#define RESOURCE_ID    127
/* Start index into the menu array */

#define ABOUT_BOX      1
/* About box item # in Apple menu */

#define I_QUIT         1
/* Quit item # in File menu */

/* Various constants */
#define NIL                    0L
#define FALSE                  false
#define TRUE                   true

/* Coords for disk init dialog box */
#define INIT_X                 112
#define INIT_Y                 80

#define APPEND_MENU            0
#define CHAR_CODE_MASK         255
#define DEFAULT_VOL            0
#define IN_FRONT               (-1)
#define MAX_TRIES              6
#define NO_CURSOR              0L
#define LONG_NAP               60L
#define SYSTEM_7               0x0700
#define FILE_SHARING_CREATOR   'hhgg'
#define FILE_SHARING_TYPE      'INIT'

/* Resource IDs for the windows & dialogs */
#define ABOUT_BOX_ID           128
#define ERROR_BOX_ID           130
#define ERROR_MESS_ID          131

/* Resource ID for the message strings */
#define LOG_ID_STR                128
#define PROBLEM_STOPPING_FS       1
/* ID numbers of the messages */
#define PROBLEM_STARTING_FS       2
#define PROBLEM_ON_EJECT          3
#define DONT_EJECT_STARTUP_VOL    4
#define CANT_FIND_STARTUP_VOL     5
```

```
#define TROUBLE_WITH_SYS_INFO    6
#define CANT_LOCATE_FILE         7
#define PROBLEM_WITH_AE_HANDLER  8
#define SYSTEM_7_REQUIRED        9

/* Bit 9 in vMAttrib field = volume is shared */
#define PERSONAL_ACCESS_MASK     0x00000200L

/* Send a message to file server */
#define SEND_MESSAGE             13
/* csCode to shut down server */
#define SHUT_DOWN                2
```

These declarations are from "SonOMunger.c" because, as stated earlier, you are reusing a lot of that code. Most Mac programmers keep handy a working "shell" that implements basic application components, such as Toolbox initialization, the event loop, simple menu functions, and high-level event handlers. Writing a new program thus becomes a matter of fleshing out the details with application-specific custom functions. This also simplifies debugging, because you're building on a proven code foundation. Now declare the functions you plan to use:

```
/* Function prototypes */
Boolean Check_System(void);
/* Standard application functions */
Boolean Do_Command (long mResult);
Boolean Init_Mac(void);
void Main_Event_Loop(void);
void Report_Error(OSErr errorCode);
void Report_Err_Message(long messageID);

Boolean Init_AE_Events(void);
/* High level Apple Events */
void Do_High_Level(EventRecord *AERecord);
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein,
                                  AppleEvent *reply,
                                  long refIn);
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein,
                                     AppleEvent *reply,
                                     long refIn);
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein,
                                   long refIn);
```

```
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein,
                                  AppleEvent *reply,
                                  long refIn);

/* Functions to handle details of file sharing */
Boolean File_Share_On(short vRefNum);
Boolean Find_File_Sharing(void);
Boolean Get_FS_Info(void);
void Stop_File_Sharing(void);
void Start_File_Sharing(void);
void Toggle_File_Sharing(void);
```

As you can see, your functions are broken down into the "generic" ones you always reuse, plus the application-specific ones. Now for some data structures:

```
/* Assorted structures for server trap */

typedef long   *LongIntPtr;

#if defined(powerc) || defined (__powerc)
#pragma options align=mac68k
#endif

struct DisconnectParam
   {
   QElemPtr    qLink;
   short       qType;
   short       ioTrap;
   Ptr         ioCmdAddr;
   ProcPtr     ioCompletion;
   OSErr       ioResult;
   LongIntPtr  scDiscArrayPtr;
   short       scArrayCount;
   short       reserved;
   short       scCode;
   short       scNumMinutes;
   short       scFlags;
   StringPtr   scMessagePtr;
   };

#if defined(powerc) || defined(__powerc)
#pragma options align=reset
#endif
```

```
typedef struct DisconnectParam DisconnectParam;
typedef union SCParamBlockRec SCParamBlockRec;
typedef SCParamBlockRec *SCParamBlockPtr;

/* Structure for adding handlers into AE event dispatch table */
struct AEinstalls
   {
    AEEventClass theClass;
    AEEventID theEvent;
    AEEventHandlerProcPtr theProc;
   };
typedef struct AEinstalls AEinstalls;


/* Globals - standard */
WindowPtr            geventWindow;
/* your private window */
EventRecord          gmyEvent;
CursHandle           gtheCursor;
/* Current pointer icon */
Boolean              guserDone;
WindowPtr            gwhichWindow;
short                gwindowCode;

/* Application-specific globals */
short                gdragNDropFlag;
ProcessInfoRec       gprocess;
ProcessSerialNumber  gprocessSN;
long                 gSysDirID;
short                gsysVRefNum;
FSSpec               gthisFileSpec;
FSSpecPtr            gthisFileSpecPtr;
```

Define your usual gaggle of globals here. The high-level event structure for the dispatch table is recycled from "SonOMunger." The other data structures are used to set up a function that controls the file server software that's at the core of File Sharing. This code illustrates an important point. Note the use of the `#pragma options align=mac68k` statement. There's a reason that it's here, due to how different processors access data in different ways.

The 680x0 processor readily accesses data bytes at any memory address. Larger data types, such as long words, must be aligned on even memory (2-byte) addresses, or an exception occurs. Put another way, the 680x0 processor requires that most data be aligned on word (16-bit) boundaries. A 680x0 compiler typically adds padding bytes at certain points in a program's data structures to ensure that they are word-aligned.

The PowerPC processor, on the other hand, favors memory alignment that conforms to the data's size. In other words, it can readily access bytes at any address, words (16 bits) at every even address, and longs (32 bits) at every address divisible by four. Note the use of the verb favors here: The PowerPC can actually access data of any size at any address. However, aligned memory fetches require fewer bus cycles than unaligned ones. This situation is summarized in Table 5.1. In special situations, misaligned data can actually trigger an exception. To minimize bus cycles and thus improve performance, PowerPC compilers insert padding bytes into data structures to achieve the preferred data alignment.

Table 5.1 Processor-Specific Data Element Alignments

Processor	*Data Element Size*	*Preferred Alignment*
680x0	byte	None
	word	2-byte alignment required
	long	2-byte alignment required
PowerPC	byte	None
	word	2-byte alignment preferred
	long	4-byte alignment preferred

Important
For the PowerPC 603, 604, and 604e, take care to ensure that elements within data structures and the contents of large blocks of data stay aligned on 4-byte (long) boundaries for optimum performance. Typically, a compiler adds padding bytes to some of the variable's storage locations to optimally align all of the variable's memory addresses to a 4-byte boundary. For complex data structures, however, with a mix of different-sized variables, misalignment can still occur and a program's performance suffers. This is one of those situations where it's up to you, the programmer, to ensure that data misalignment doesn't occur. See Chapter 7 for additional information.

A data structure that's optimally aligned for the PowerPC processor might not be word-aligned and thus not usable by a 680x0 processor. On a Power Macintosh, you might wonder why you would care about data alignment anyway. Recall that the Mac Toolbox still implements many routines as 680x0 code, and the 68LC040 emulator expects the data to be word aligned. Also, there are still plenty of 680x0-based Macs out there that your software should support. For example, suppose your program creates files with internal data structures that you expect a 680x0 Mac to read. Likewise, a networked Power Mac might transfer data through the network to 680x0 Macs for use. In both cases, proper data alignment is crucial.

To avoid this problem, the `#pragma options align=mac68k` statement tells the compiler to word-align the program's data structures. Performance may suffer on a PowerPC, but this data arrangement ensures that the 680x0 processor accesses will operate, especially for emulated Toolbox code. The `align=reset` directive immediately after `DisconnectParam` structure tells the compiler to resume arranging data in the PowerPC's preferred data alignment scheme. This is done to minimize the impact of misaligned data accesses on the PowerPC processor. The header files handle most of these alignment issues for you, much the same way that they set up the UPPs for certain Toolbox calls. However, for the custom function here—or any data structure you expect to pass the Mac OS or 680x0 Mac—you have to take care of the alignment problem yourself.

Now it's time to write your custom function:

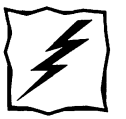
```
/* Glue to call the ServerDispatch trap */
#if USES68KINLINES
#pragma parameter __D0 mySyncServerDispatch(__A0)
#endif
pascal OSErr mySyncServerDispatch(SCParamBlockPtr PBPtr)
 FOURWORDINLINE(0x7000, 0xA094, 0x3028, 0x0010);

/* = {   */
/* 0x7000, /* MOVEQ  #$00, D0     ; Input must be 0*/
/* 0xA094, /* _ServerDispatch   ; Hop to the trap*/
/* 0x3028,*/
/* 0x0010 /*
MOVE.W ioResult(A0),D0; Move result to D0 because*/
/* }        ; File Sharing doesn't.*/


#ifdef powerc
// Call the 680x0 code from the PowerPC through Mixed Mode Manager
static pascal OSErr mySyncServerDispatch(SCParamBlockPtr PBPtr)
{
ProcInfoType myProcInfo;
OSErr result;
   /* Need an RTS at the end to return ... */
static short code[] = {0x7000, 0xA094, 0x3028, 0x0010, 0x4E75};

   /* Build the procinfo (note use of register based calls) */
   myProcInfo = kRegisterBased
                      ¦ RESULT_SIZE(SIZE_CODE(sizeof(OSErr)))
                      ¦ REGISTER_RESULT_LOCATION(kRegisterD0)
                      ¦ REGISTER_ROUTINE_PARAMETER(1,kRegisterA0,
                        SIZE_CODE(sizeof(SCParamBlockPtr)));


   result = CallUniversalProc((UniversalProcPtr) code,
                              myProcInfo, (PBPtr));
   return result;
} /* end mySyncServerDispatch() */
#endif
```

`SyncServerDispatch()` is a Toolbox routine that controls the file server software that implements AppleShare and File Sharing. Unfortunately, this routine escaped being defined in Apple's PowerPC libraries. Because I happened to

know the trap word and glue code for SyncServerDispatch(), it seemed that I could define the routine call myself. I wrote a function called mySyncServerDispatch() that uses the 680x0 assembly language code to implement the missing routine call. The conditional flags USES68KINLINES and powerc make CodeWarrior use either the original in-line 680x0 machine code when making the 680x0 version of the application, or use the PowerPC function when making the PowerPC application. By placing the 680x0 function definition before the PowerPC's, the 680x0 function serves double-duty as the PowerPC's function prototype. All the PowerPC function does is create the appropriate routine descriptor for mySyncServerDispatch() before calling the same 680x0 machine code. Let's see how this is done.

In the PowerPC version, declare mySyncServerDispatch() as static to give its name file scope instead of global scope. You use the same in-line machine code as in the 680x0 version of the function, but with an important twist. On a 680x0 processor, the routine's code executes in-line, with execution resuming at the next instruction when the processor returns from the A trap exception. However, for the PowerPC version, you call this in-line code as a function. To return properly to the calling function, you add a 680x0 RTS instruction (0x4E75) to the code. You have to call the routine this way so that the Mixed Mode Manager can step in and handle the instruction set context switch.

Next, you construct the data structure myProcInfo, which contains a description of mySyncServerDispatch()'s arguments. This routine is register based: That is, the argument and result get passed in 680x0 processor registers. MySyncServerDispatch() expects a pointer to a parameter block that contains a server control command in register A0, and the result of the operation is returned in register D0. Once myProcInfo is set up to describe this arrangement, you call CallUniversalProc(), and pass it the address of the routine call (the first element of the array code), myProcInfo, and the parameter block with the server command. When calling a 680x0 routine, as in this example, you can see that the 680x0 code pointer is a UPP. Therefore, it's unnecessary to create a routine descriptor for a 680x0 routine. This is why CallUniversalProc() takes the routine descriptor information as a separate argument. Generating code on-the-fly in the code array this way isn't the best implementation of the routine call, but it works adequately for this particular program.



Important

Implementing a function by using a code array can create potential problems on PowerPC processors with a Harvard architecture (that is, separate instruction and data caches), such as the PowerPC 603, 603e, 604, and 604e. (The 601 uses a unified cache that can contain both code and data.) Because the machine instructions inside the array technically are data, they could wind up in the processor's data cache instead of the instruction cache, creating no end of problems for program execution. However, I've conducted tests on a PowerBook 5300 (which has a 603e) and a Power Mac 9500 (which has a 604), and this function works fine. Why? It works because the function is written in 680x0 code, which is treated as data at all times by the 680x0 emulator. This wouldn't be the case if the function were written in native code, so beware of writing native function calls this way.

You've made a change to the error reporting function:

```
void Report_Err_Message(long messageID)
{
unsigned char errorString[256];

   GetIndString((unsigned char *)
         errorString, LOG_ID_STR, messageID);
   if (errorString[0] == 0)
/* Is there a string present? */
      {
      SysBeep(30);         /* No, give up */
      return;
      } /* end if */
   ParamText(errorString, NIL, NIL, NIL);
   CautionAlert(ERROR_MESS_ID, NIL);

} /* end Report_Err_Message() */
```

Instead of accepting a pointer to a Pascal string, `Report_Err_Message()` now processes a message ID number. This message ID number corresponds to the ID number of a STR# resource that contains the relevant error message.

This function uses `GetIndString()` to retrieve a string and passes it to the routines `ParamText()` and `CautionAlert()` for display. You do one safety check here. The Pascal string format has a length byte at the start of the string, and it is followed by the string data. The length byte tells how many characters are in the string. If this value is zero, something's gone awry, and you bail out.

The basic error function, `Report_Error()`, hasn't changed.

Here are the first application-specific functions:

```
Boolean Get_FS_Info(void)
{
gthisFileSpecPtr = &gthisFileSpec;
gprocessSN.highLongOfPSN = kNoProcess;
gprocessSN.lowLongOfPSN = kNoProcess;

 /* Store record size */
gprocess.processInfoLength = sizeof(ProcessInfoRec);
 /* Allocate room forthe name */
gprocess.processName = (unsigned char *) NewPtr(32);
/* Direct towards your storage */
gprocess.processAppSpec = gthisFileSpecPtr;
/* Loop until all processes found */
while (GetNextProcess(&gprocessSN) == noErr)
    {
    if (GetProcessInformation(&gprocessSN, &gprocess) == noErr)
        {
        /* Is this process the File Sharing Extension? */
        if (gprocess.processType == FILE_SHARING_TYPE &&
         gprocess.processSignature == FILE_SHARING_CREATOR)
           return TRUE;
         } /* end if */
    } /* end while */

return FALSE;
}/* end Get_FS_Info() */
```

`Get_FS_Info()` searches the Mac OS process list, looking for a process whose file signature is the File Sharing Extension. You'll notice that you swiped most of this code from "process.c." This function assumes File Sharing is active, which means its process is present. If you discover such a process,

`Get_FS_Info()` returns TRUE. The file name connected to this process is saved in the global `gprocess.processAppSpec`. If the process list is walked without finding a match, it returns FALSE.

```
Boolean File_Share_On(short volRefNum)
{
HParamBlockRec    ioHPB, volHPB;
GetVolParmsInfoBuffer   volInfoBuffer;

/* Get volume reference number */
   volHPB.volumeParam.ioCompletion = NIL;   volHPB.volumeParam.ioNamePtr = NIL;
/* No volume name */
   volHPB.volumeParam.ioVRefNum = volRefNum;
/* 0 = Use only volRefNum to obtain the info */
   volHPB.volumeParam.ioVolIndex = 0;
   if (!PBHGetVInfo(&volHPB, FALSE))
      {
/* Get volume's characteristics */
      ioHPB.ioParam.ioCompletion = NIL;
      ioHPB.ioParam.ioNamePtr = NIL;
      ioHPB.ioParam.ioVRefNum = volHPB.volumeParam.ioVRefNum;
      ioHPB.ioParam.ioBuffer = (char *) &volInfoBuffer;
      ioHPB.ioParam.ioReqCount = sizeof(volInfoBuffer);
      if (!PBHGetVolParms(&ioHPB, FALSE))
         {
         if (volInfoBuffer.vMAttrib & PERSONAL_ACCESS_MASK)
            { /* The disk is shared */
            if (Get_FS_Info())
/* Look for File Sharing Ext */
               return TRUE;
/* Got the file info you need */
            } /* end if */
         } /* end if !PBHGetVolParms */
      } /* end if !PBHGetVInfo */
   return FALSE;
} /* end File_Share_On() */
```

This is part of the program's design to encourage users to drop all their volume icons onto the SwitchBank application. `File_Share_On()` is used to determine whether the volume dropped onto SwitchBank is shared. If `File_Share_On()` reports that the volume isn't shared, SwitchBank will eject it without interrupting File Sharing. The idea is to avoid excessive stopping and starting of the File Sharing process, which can fragment memory.

The routine `PBHGetVInfo()` takes the volume specification supplied to it via `volRefNum` and converts it to a volume reference number. This value, which is returned in `volHPB.volumeParam.ioVRefNum`, is used in the routine `PBHGetVolParms()` to obtain information about the target volume. If bit 9 in the `vMAttrib` field is set, the volume is being shared. You confirm this by calling `Get_FS_Info()`, which checks for the File Sharing Extension process. This also primes your global `gprocess.processAppSpec` with the file information associated with the process.

With these functions, you have obtained enough information about the File Sharing process to switch it off or on. Starting with shutting it off, you have:

```
/* Send a shut down immediately message */
/* to the File Sharing Server */
void Stop_File_Sharing(void)
{
DisconnectParam   serverBlock;
SCParamBlockPtr   serverBlockPtr;

/* Point to the message block */
   serverBlockPtr = (SCParamBlockPtr) &serverBlock;
   serverBlock.scCode = SHUT_DOWN;
/* Server cmd to shut down */
   serverBlock.scNumMinutes = 0;
/* Do it immediately */
   serverBlock.scFlags = SEND_MESSAGE;
   serverBlock.scMessagePtr = NIL;

   if (mySyncServerDispatch(serverBlockPtr) == noErr)
      {
      /* Let the OS get at the event */
      WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
      WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
      } /* end if */
      else
         Report_Err_Message(PROBLEM_STOPPING_FS);
} /* end Stop_File_Sharing() */
```

The function `Stop_File_Sharing()` does exactly what it describes. It accomplishes this by first loading the appropriate values into a parameter block that forms a server shutdown command with no time delay interval. Then it calls your custom function `mySyncServerDispatch()` with this parameter block. This issues the server shutdown command to the Mac OS.

What `Stop_File_Sharing()` executes next might not seem obvious, but it's part of the reality of cooperative multitasking. If SwitchBank plowed inexorably onward, the shutdown command wouldn't take effect. That's because SwitchBank must surrender processor time so that the shutdown command percolates through the operating system, and for the File Sharing Extension process to respond to this command. Therefore, you call `WaitNextEvent()` to give processor time to the operating system and other processes. To ensure that this occurs even when the Mac is operating under a heavy load, call it twice. If for some reason `mySyncServerDispatch()` reports an error, display an error message that explains the problem.

```
/* Launch the file that has the File Sharing */
/* application in it.  The file name used for */
/* the launch was obtained from the process */
/* when it's memory, or by searching the */
/* startup disk. */
void Start_File_Sharing(void)
{
OSErr                launchErr;
LaunchPBPtr           thisAppPBPtr;
LaunchParamBlockRec   thisAppParams;

   gthisFileSpecPtr = &gthisFileSpec;
   thisAppPBPtr = &thisAppParams;
   thisAppParams.launchBlockID = extendedBlock;
// Use new format
   thisAppParams.launchEPBLength = extendedBlockLen;
   thisAppParams.launchFileFlags = 0;
// Don't care about flags
   thisAppParams.launchControlFlags = (launchNoFileFlags +
                                       launchContinue +
                                       launchDontSwitch);
/* Give it file name grabbed by Get_FS_Info() before */
/* File Sharing Sharing was stopped */
   thisAppParams.launchAppSpec = gthisFileSpecPtr;
   thisAppParams.launchAppParameters = NIL;
/* Send Open event */
   if ((launchErr = LaunchApplication(thisAppPBPtr)) == noErr)
      WaitNextEvent(everyEvent, &gmyEvent, SHORT_NAP, NO_CURSOR);
   else
      Report_Err_Message(PROBLEM_STARTING_FS);
} /* end Start_File_Sharing() */
```

`Start_File_Sharing()` undoes the work of `Stop_File_Sharing()`. First, it takes the file name that it obtained from `Get_FS_Info()` or `Find_File_Sharing()`, (described later) and puts it in a parameter block. You also set control flags in this parameter block that specify your application should continue running after the target application launches, and for the target application (File Sharing Extension) not to switch to the foreground. `Start_File_Sharing()` then calls the `LaunchApplication()` routine to start the application embedded in the File Sharing Extension file. Again, you have to call `WaitNextEvent()` so the operating system gets an opportunity to handle the command.

Important

The initial version of SwitchBank sent a high-level Quit Application Apple Event that stopped the File Sharing Extension process. However, versions of File Sharing prior to System 7.5.1 had problems restoring server connections if the process were halted and resumed on the fly. The proper way to stop this process was through file server commands, because File Sharing operates as a file server. Also, using the server command provides a terrific example of how to set up and use a routine descriptor.

For those who are interested in how to halt File Sharing using an Apple Event, here's the original `Stop_File_Sharing()` function. The code illustrates how to package a Quit Application Apple Event and send it to another application.

```
OSErr Stop_File_Sharing(void)
{
OSErr       err;
AppleEvent  thisEvent;
AEDesc      thisAddress;

if (File_Share_On()) /* Turn it off */
   {
   err = AECreateDesc(typeProcessSerialNumber, &gprocessSN,
                      sizeof(processSN), &thisAddress);
   if (!err)
      err = AECreateAppleEvent(kCoreEventClass,
                               kAEQuitApplication,
                               &thisAddress,
```

```
                                kAutoGenerateReturnID,
                                kAnyTransactionID, &thisEvent);
   if (!err)
      err = AESend(&thisEvent, nil, kAENoReply +
                    kAEAlwaysInteract +
                    kAECanSwitchLayer, kAENormalPriority,
                    kAEDefaultTimeout, nil, nil);
   if (!err)
      {
      AEDisposeDesc(&thisAddress);
      AEDisposeDesc(&thisEvent);
      } /* end if */
   /* Let the OS handle the event */
   WaitNextEvent(everyEvent, &myEvent, LONG_NAP, NO_CURSOR);
   } /* end if fileShareOn */

return err;
} /* end Stop_File_Sharing() */
```

The SwitchBank Controls menu lets you switch on or off File Sharing with just a keystroke. However, there's a problem here: What if the user started his Macintosh with File Sharing off, and wants to turn it on? In this situation, there's no File Sharing process running in memory that `Get_FS_Info()` can retrieve a file name from for `LaunchApplication()` to use. To close the door on this potential pitfall, I wrote the `Find_File_Sharing()` function:

```
Boolean Find_File_Sharing(void)
{
HParamBlockRec    searchPB;
FInfo             fileSharingExtInfo, fileSharingMaskInfo;
CInfoPBRec        searchSpec1, searchSpec2;
Point             nilPoint = {0, 0};

/* Set up creator and type for File Sharing Extension */
    fileSharingExtInfo.fdType = FILE_SHARING_TYPE;
    fileSharingExtInfo.fdCreator = FILE_SHARING_CREATOR;
    fileSharingExtInfo.fdFlags = 0;
    fileSharingExtInfo.fdLocation = nilPoint;
    fileSharingExtInfo.fdFldr = 0;
```

```
/* Set up masks */
    fileSharingMaskInfo.fdType = (OSType) 0xffffffff;
    fileSharingMaskInfo.fdCreator = (OSType) 0xffffffff;
    fileSharingMaskInfo.fdFlags = 0;
    fileSharingMaskInfo.fdLocation = nilPoint;
    fileSharingMaskInfo.fdFldr = 0;

/* 1st spec block */
/* Search by file type, not name */
   searchSpec1.hFileInfo.ioNamePtr = NIL;
/* Type & creator to look for */
   searchSpec1.hFileInfo.ioFlFndrInfo = fileSharingExtInfo;

/* 2nd spec block */
   searchSpec2.hFileInfo.ioNamePtr = NIL;
   searchSpec2.hFileInfo.ioFlFndrInfo = fileSharingMaskInfo;
/* Set up search call */
   searchPB.csParam.ioCompletion = NIL;
   searchPB.csParam.ioNamePtr = NIL;
/* No volume name */
/* Search on startup volume */
   searchPB.csParam.ioVRefNum = gsysVRefNum;
/* Result goes here */
   searchPB.csParam.ioMatchPtr = &gthisFileSpec;
      searchPB.csParam.ioReqMatchCount = 1;
/* Look for 1 file */
/* Search based on file characteristics */
   searchPB.csParam.ioSearchBits = fsSBFlFndrInfo;
   searchPB.csParam.ioSearchInfo1 = &searchSpec1;
   searchPB.csParam.ioSearchInfo2 = &searchSpec2;
   searchPB.csParam.ioSearchTime = 0;
/* Don't time out */
/* Start at the beginning */
   searchPB.csParam.ioCatPosition.initialize = 0;
/* No search cache required */
   searchPB.csParam.ioOptBuffer = NIL;
       searchPB.csParam.ioOptBufSize = 0;

   if (PBCatSearchSync((CSParamPtr) &searchPB) == noErr)
      return TRUE;
   else
      {
```

```
        Report_Err_Message(CANT_LOCATE_FILE);
        return FALSE;
        } /* end else */
} /* end Find_File_Sharing() */
```

In this function, you search for the File Sharing Extension on the startup volume, or boot disk. You begin by setting up the file's signature information (its creator and type) in a `fileSharingExtInfo` structure. This signature information is what you give the routine `PBCatSearchSync()` so that it can locate the file. The `PBCatSearchSync()` routine performs high-speed searches on a volume's catalog file for specific file or directory information, and is ideal for the job. For more information, consult *Inside Macintosh: Files*. `PBCatSearchSync()` requires two specification blocks. The first contains search information and a start range, the second contains any masks to filter out information and a stop range. Your mask information, in the `fileSharingMaskInfo` structure, passes only the file's creator and type. Next, you assemble the parameter block that furnishes `PBCatSearchSync()` with the information needed to conduct the search. You supply it a pointer to the file specification global, `gthisFileSpec`, for the result to land in. You also provide the volume reference number of the startup volume so that the search is conducted on the volume that has the System Folder. This is done because the File Sharing Extension resides in the Extensions folder, which in turn lies in the System Folder. If `PBCatSearchSync()` returns with a match, the global `gthisFileSpec` contains the file name, which is ready for use in `Start_File_Sharing()`.

Background Info

You might be wondering why `Get_FS_Info()` and `Find_File_Sharing()` both use the File Sharing Extension's signature information when they search, rather than just simply plugging in the name "File Sharing Extension." If you used a file name instead, it hampers the program's capability to operate in Macs overseas. That's because the File Sharing Extension's name varies in different languages, while its file signature data never changes.

The next function implements the File Sharing toggle function used in SwitchBank's Controls menu. It's pretty simple, and it just calls the other functions discussed previously.

```
void Toggle_File_Sharing(void)
{
   if (Get_FS_Info()) /* File Sharing already on (& in memory)? */
      Stop_File_Sharing(); /* Yes, turn it off */
   else     /* No, look for the file */
      {
      if (Find_File_Sharing())
/* Find the File Sharing Extension file */
         Start_File_Sharing();
/* Launch it */
      } /* end else */
} /* end Toggle_File_Sharing() */
```

Some of the functions in SwitchBank, such as the one that installs the core Apple Event handlers, haven't changed and won't be covered here. For a complete source code listing, check the file "SwitchBank.c" on the CD-ROM, or Appendix C. However, what has changed is the new Open Document handler, as shown here:

```
/* High-level open document event */
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein,
                                     AppleEvent *reply,
                                     long refIn)
{
long               dummyResult;
/* Dummy variable for delay() */
register short      i, j;
Boolean            fileShareWasOn;
AEDesc             volDesc;
/* Container for sent volume names */
OSErr              volErr, highLevelErr;
/* Number of volumes dropped onto you */
long               numberOVolumes;
/* Bit buckets for high-level event info */
AEKeyword          ignoredKeyWord;
DescType           ignoredType;
Size               ignoredSize;
 /* Container for volume names as FSSPecs */
FSSpec             volFSS;

   gtheCursor = GetCursor(watchCursor);
/* Change cursor */
   SetCursor(&**gtheCursor);
```

```
   fileShareWasOn = FALSE;

   if (!(highLevelErr = AEGetParamDesc(messagein, keyDirectObject,
                                    typeAEList, &volDesc)))
      {
      if ((highLevelErr = AECountItems(&volDesc, &numberOVolumes))
                == noErr)  /* How many? */
         {
         for (i = 1; ((i <= numberOVolumes) &&
            (!highLevelErr)); ++i)
/* Process each vol */
            {
            if (!(highLevelErr = AEGetNthPtr(&volDesc, i,
                                       typeFSS,
                                       &ignoredKeyWord,
                                       &ignoredType,
                                       (char *)&volFSS,
                                       sizeof(volFSS),
                                       &ignoredSize)))
               {
               if (volFSS.vRefNum != gsysVRefNum)
                 /* Chosen volume the boot drive? */
                  {
                  if (File_Share_On(volFSS.vRefNum))
                 /* This volume being shared? */
                     {
                     Stop_File_Sharing();
                     fileShareWasOn = TRUE;
                     } /* end if */
                  j = 0;   /* Set retry count */
                  while (((volErr = Eject(volFSS.name,
                                    volFSS.vRefNum)) !=
                                    noErr) &&
                                    (j < MAX_TRIES))
                     {
                     WaitNextEvent(everyEvent,
                                   &gmyEvent, SHORT_NAP,
                                   NO_CURSOR);
                     Delay(10L, &dummyResult);
                     j++;
                     } /* end while */
                  if (volErr == noErr)
                     UnmountVol(volFSS.name,
```

```
                                volFSS.vRefNum);
                    else
                        Report_Err_Message(PROBLEM_ON_EJECT);
                    } /* end if != gsysVRefNum */
                else
                    Report_Err_Message(DONT_EJECT_STARTUP_VOL);
                } /* end if !highLevelErr */
            } /* end for */
        } /* end if */
          /* Release memory copy of the AE parameter */
      highLevelErr = AEDisposeDesc(&volDesc);
      } /* end if !highLevelErr */

   if (fileShareWasOn)
      Start_File_Sharing();

   if (gdragNDropFlag >= 0)  /* Did user drag & drop? */
      guserDone = TRUE;      /* Yes, stop the application */
   SetCursor(&qd.arrow);     /* Restore the cursor */
   return highLevelErr;      /* Kick back any high-level
                                problems to calling app */
} /* end Core_AE_OpenDoc_Handler() */
```

This function, like in SonOMunger, gets called when objects get dropped on the application's icon. The handler uses the routine `AEGetParamDesc()` to fetch the message parameter out of the Open Document event sent to it and coerces the data into a descriptor list. The `AECountItems()` routine extracts the number of items in the list, and this value sets the duration of a `for` loop. The `for` loop uses the routine `AEGetNthPtr()` to obtain each volume name from the descriptor list, converting it into a file system specification as it does so. As in SonOMunger, if `AEGetNthPtr()` reports an error, the loop stops. The target volume is now subjected to a battery of tests. First, its reference number is checked against the startup volume's reference number. If the two match, the user is attempting to eject the drive with the system software on it, which is a very bad idea. SwitchBank thus intervenes and warns the user of this. You could ignore this problem and allow the operation to fail when, because of the system software, the Toolbox `Eject()` routine detects the volume is busy. However, you can supply the user with a more informative error message and save some wasted processor cycles if you do this check now.

Next you look to see whether the volume is being shared. If so, you call `Stop_File_Sharing()` to turn off File Sharing and set the flag `fileShareWasOn` to remind you that you did so. At last you call `Eject()` to eject the volume. The

nice thing about this Toolbox routine is that it handles any type of volume: CD-ROM, floppy, and networked hard drives. If the Mac is extremely busy, `Eject()` might report an error because File Sharing hasn't had a chance to stop yet. So you wait one-sixth of a second, call `WaitNextEvent()`, and retry the operation. The value of `MAX_TRIES` for this loop was determined empirically—a way of saying that I used SwitchBank on a Power Mac running under a heavy load and experimented until I found a value that worked best. If the eject fails, warn the user.

When all of the work is done, clean up by first disposing of the copy of the descriptor list made by `AEGetParamDesc()`. Then check `fileShareWasOn` to see whether File Sharing needs to be restarted, and call `Start_File_Sharing()` as required. Finally, check a flag called `gdragNDropFlag` to determine whether the application is already running, or was launched because of a drag-and-drop action. If the latter, flip the state of `guserDone` to make SwitchBank quit. If any of the high-level Apple Event routines have reported an error, pass this value back to the calling application.

A new function in your stable of shell routines is `Check_System()`. As its name implies, it's used to check for specific features the application might need to operate properly:

```
Boolean Check_System(void)
{
SysEnvRec   machineInfo;
/* Record with machine-specific data */
short       sysVersion;
/* System version # */
/* Version of SysEnvirons() to use */
short       versionRequested;

      sysVersion = SYSTEM_7;

/* MUST set this to get valid results */
      versionRequested = 1;

      if (SysEnvirons(versionRequested, &machineInfo) == noErr)
            sysVersion = machineInfo.systemVersion;
      else
            {
            Report_Err_Message(TROUBLE_WITH_SYS_INFO);
            return FALSE;
```

```
            } /* end else */

      if (sysVersion < SYSTEM_7)
/* Running System 7.0? */
            {
            Report_Err_Message (SYSTEM_7_REQUIRED);
            return FALSE;
/* No. Sorry, can't run without it  */
            } /* end if */

      return TRUE;
} /* end Check_System() */
```

The preferred method for investigating certain system features is to use the Gestalt Manager. However, the Gestalt Manager is available only under System 6.0.5 or later, which can be a problem if someone happens to launch SwitchBank on a Mac running an earlier version of the Mac OS. Because SwitchBank relies so heavily on certain System 7 features such as Apple Events, File Sharing, and the catalog search performed by `Find_File_Sharing()`, you must check the operating system version number. The solution is to call an older routine, `SysEnvirons()`. You use it to return the operating system version number, which you compare to see whether it's System 7 or later. If not, `Check_System()` returns FALSE so that SwitchBank aborts the initialization phase. Once you have determined that System 7 is running, then you can use Gestalt Manager calls to look for specific features. Examples of how to use the Gestalt Manager to determine system-specific features abound in every volume of the new editions of *Inside Macintosh*, and volume VI of the old editions of *Inside Macintosh*. In this case, the check for the presence of System 7 is sufficient.

In `Do_Command()` you add an entry for SwitchBank's Controls menu, like so:

```
            case EDIT_MENU:
                  SystemEdit(theItem - 1);
            break;

            case SWITCH_MENU:
                  Toggle_File_Sharing();
            break;

            default:
            break;
```

There are also some minor changes to `Main_Event_Loop()`. At the start of the function, you set `gdragNDropFlag` equal to 1. At the end of the do

```
            } /* end switch gmyEvent.what */
         } /* end if on next event */
      else                          /* Null event */
           ;  /* Do idle or background stuff here */

/* Flag to determine whether app was launched by */
/* user or Open Apple Event */
       if (gdragNDropFlag >= 0)
          gdragNDropFlag--;
       } /* end do */

   while (guserDone == FALSE)
      ;              /* Loop until told to stop */
} /* end Main_Event_Loop() */
```

Here's where the flag `gdragNDropFlag` gets set. For a drag-and-drop operation, execution passes through the event loop twice (once for the Open Application Apple Event, and once for the Open Document Apple Event). The event loop decrements `gdragNDropFlag` until it goes negative, which occurs if the loop is traversed three times or more. At this point, if the value is negative, you can safely assume the application was launched by the user, and so `gdragNDropFlag` prevents `Core_AE_OpenDoc_Handler()` from stopping the application if a volume is dragged onto SwitchBank.

The initialization function has changed:

```
Boolean Init_Mac(void)
{
Handle      theMenuBar;

/* Lunge after all the memory you can get */
   MaxApplZone();

/* Make sure you've got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
```

```
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(NIL);

/* Got the menu resources OK? */
   if ((theMenuBar = GetNewMBar(MENU_BAR_ID)) == NIL)
      return FALSE;

   SetMenuBar(theMenuBar);
/* Add your menus to menu list */
   DisposHandle(theMenuBar);

 /* Make Apple menu */
   AppendResMenu(GetMenuHandle(APPLE_MENU), 'DRVR');

   DrawMenuBar();

/* Look for specific features or set up */
/* handlers this app needs */
   if (!Check_System())  /* Need System 7 */
      return FALSE;

   if (!Init_AE_Events())
/* Set up high-level event handlers */
      return FALSE;

   if (FindFolder(kOnSystemDisk, kSystemFolderType,
         kDontCreateFolder, &gsysVRefNum, &gSysDirID) != noErr)
      {
      Report_Err_Message (CANT_FIND_STARTUP_VOL);
      return FALSE;
      } /* end if */
```

```
    InitCursor();  /* Tell user app is ready */
    return TRUE;
} /* end Init_Mac() */
```

This time you build your menus by using `GetNewMBar()` and the MBAR resource you made in SwitchBank.r. This eliminates the array of MenuHandles and the `for` loop you used in SonOMunger. However, if you want to add hierarchal menus in your application, you'll still have to use the `InsertMenu()` routine to set them up. You call `Check_System()` to see whether the Mac is running System 7, followed by `Init_AE_Events()` to install your high-level event handlers. Finally, you call `FindFolder()`, a routine that obtains information on system directories such as the Preferences folder. If you pass this routine the constants `kOnSystemDisk` and `kSystemFolderType`, you get the startup volume reference number. From the earlier function descriptions, you'll recall that you needed this information for some error checking and the catalog search. The `main()` function hasn't changed at all. To examine the complete source code of the program, open the SwitchBank.c file in the SwitchBank folder, or check Appendix C.

Creating the SwitchBank application uses the standard make operation you performed on SonOMunger. Compile the SwitchBank.c file, and correct any errors. Ensure that only the libraries "MCWRuntime.Lib" and "Interface.Lib" are in the project. Then, go to the Edit menu and select Preferences. Choose the PPC Project panel and change the application's name to SwitchBank, the creator to SWCH, and the memory size to 384 K (see Figure 5.9). Click on the SIZE flags popup menu here and uncheck the items `acceptSuspendResumeEvents`, `doesActivateOnFGSwitch`, and `canBackGround`. The first two flags indicate that an application does its own window maintenance when the application gets switched from the foreground to the background, or vice-versa. Because SwitchBank doesn't have any windows other than a modal dialog box for its About window, disabling these flags tells the Mac OS not to bother sending it any Suspend/Resume events. Finally, because SwitchBank doesn't do any background processing, there's no reason for the Mac OS to send it null events, which is why you disable the `canBackGround` flag. This helps balance the load for those applications that do perform background processing and need null events. Confirm that the flag `isHighLevelEventAware` is checked on this menu. Now, make the application, and remember to rebuild the desktop database. The result should be an application with a knife-switch icon. You can launch the application, and

from the Controls menu toggle File Sharing on or off. If you leave the application on the Desktop, you can drag and drop any volume icon onto SwitchBank. SwitchBank automatically launches, stops File Sharing if required, ejects the volume, restarts File Sharing, and quits.

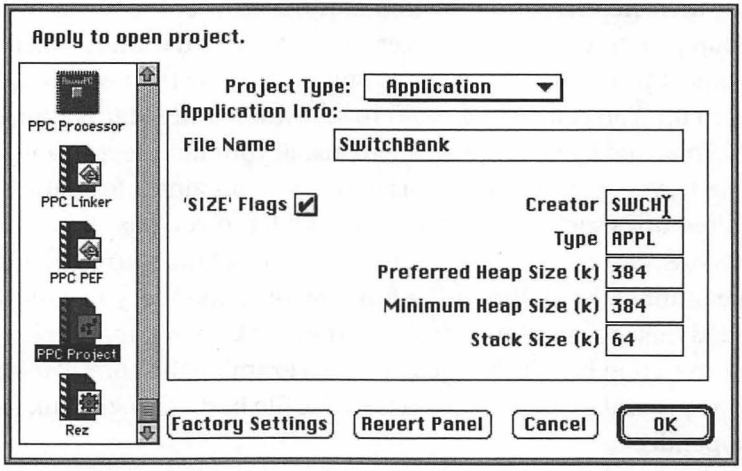


Figure 5.9 Adjusting the project settings for the SwitchBank application.

The end result is a small utility application that makes my life easier. It also taught me a lot about the Power Mac's operating system. With the release of System 7.5.1, Apple modified the behavior of File Sharing so that it no longer holds a shared CD-ROM captive. Fine and well, but many Power Macs still run System 7.1.2 or System 7.5, so SwitchBank is still of value to some users. For the gentle reader, it's a good example of how to write a custom function and its routine descriptor. You should also have a better picture of how the Mac OS works.

Making a Fat Binary

You know I think that you should support 680x0-based Macs, if only for the simple reason there are millions of them out there. For the next several years or so, count on them to outnumber the Power Macs, although the installed

base of Power Macs is growing rapidly. It makes sense to support this large existing hardware base with your software. However, you might be wondering how you're going to maintain and manage two copies of your application, one for each type of Mac. The first issue, maintenance, is simple. If you write the C code carefully, one set of source files can be used to generate both 680x0 and PowerPC machine code (or binaries). In fact, all of the programs presented in this book can be compiled without modification in the CodeWarrior IDE using either the 680x0 or PowerPC processor as the target.

The second issue appears to be more serious. How do you ensure that a 680x0 Mac owner gets a 680x0 version of your application and not the PowerPC version? The answer is that Apple's PowerPC application design enables you to create one copy of an application that runs on both a 680x0 Mac and a Power Mac. As you'll recall in Chapter 4, a PowerPC application's code resides in a file's data fork, while a 680x0 application's code is composed of CODE resources in the file's resource fork. Both applications use a common set of resources such as MENU, WIND, DLOG, and others to implement the user interface. Because each program's code is in a different file fork, yet they draw on the same graphical resources, it's possible to make what's known as a "fat binary," as shown in Figure 5.10. Now each version of the Mac OS sees what it expects: the 680x0 Mac OS finds CODE resources in the application's resource fork, and the PowerPC Mac OS finds PowerPC code in the application's data fork. (Note: The Process Manager won't look for code fragments in an application unless a cfrg resource is present.) Due to smart planning on Apple's part, the issue of managing two different versions of the same application goes away, because one version will suffice. There are exceptions where it makes better sense to support two copies of the application. One case might be where the application is a large file, say, several megabytes. Making this application into a fat binary can double the file's size, resulting in a box of floppies and a lengthy installation for the user. In this situation, separate application binaries would keep the installation job to a manageable size and reduce the application's footprint on the system.

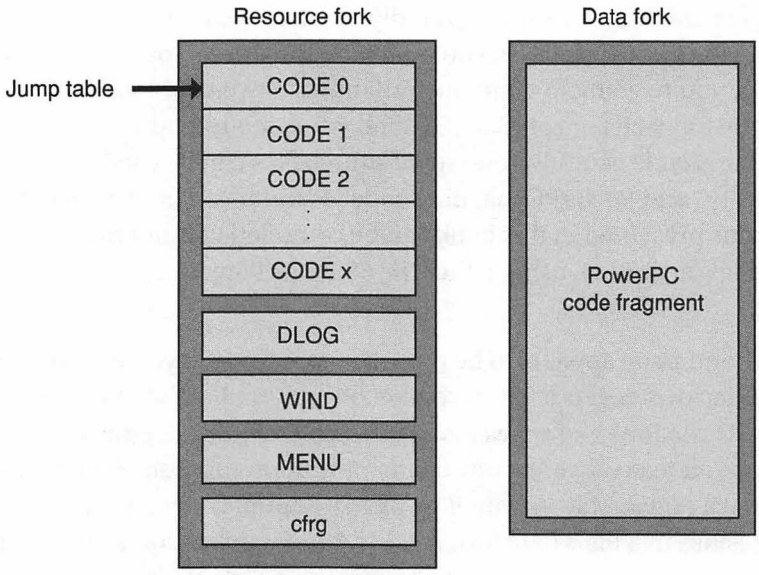


Figure 5.10 The file structure for a fat binary application.

Making a fat binary with the Metrowerks CodeWarrior isn't difficult and has two stages. By way of example, let's make "SwitchBank" into a fat binary. The first stage involves making the 680x0 version of the program. To begin, construct a new project called SwitchBank.µ.68K that you'll use to build SwitchBank as a 680x0 application. When the Standard File Save dialog box appears, go to the Project Stationary item and choose MacOS 68k C/C++.µ from the popup menu. This informs the CodeWarrior IDE that you're using the 680x0 tools for this project. As a final check, go to the Target pane in the Preferences window and ensure that Macintosh 68 K is selected as the target. Remove any surplus library files from the project window, except MacOS.lib. Now add SwitchBank.c and SwitchBank.r, as shown in Figure 5.11. Before you make the application, go to the 68 K Project pane in the Preferences window and name the output file SwitchBank.rsrc. Build the project. This results in the stand-alone application, complete with the oddball name. This completes the first stage.

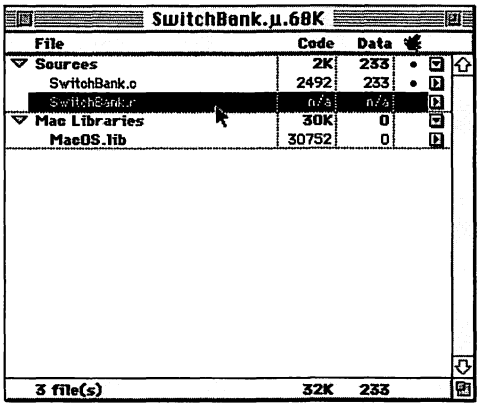


Figure 5.11 The Project window for building the 680x0 version of SwitchBank.

The second stage uses the results of the first stage, plus the output from compiling the same source code with the CodeWarrior PowerPC compiler. Start by creating a project named "SwitchBank.µ.PPC." Be sure to pick MacOS PPC C/C++.µ from the Project Stationary popup menu, and check that Macintosh PowerPC is set in the Target pane of the Preferences window. Add the SwitchBank.c file, and remove any surplus library files so that only MWCRuntime.Lib and "InterfaceLib" are in the project file. Don't bother to add the SwitchBank.r file, because you are going to use the compiled resources from the 680x0 version of SwitchBank. To do this, add the file SwitchBank.rsrc to the project, as shown in Figure 5.12.

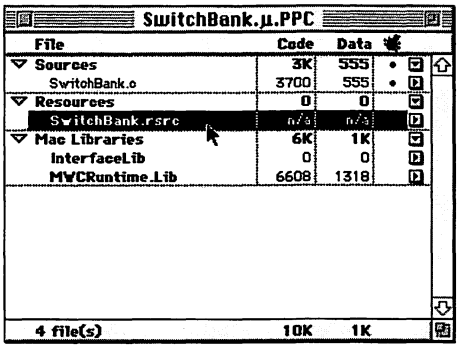


Figure 5.12 The Project window for building the SwitchBank fat binary.

Go to the Preferences item in the Edit menu and select the PPC Project panel. Set the project type to application and name the output file SwitchBank. Set the output file's type to 'APPL' and its creator to 'SWCH'. Go to the SIZE flags popup menu and check the following flag bits: `is32BitCompatible` and `isHighLevelEventAware`. Uncheck the `acceptSuspendResumeEvents`, `doesActivateOnFGSwitch`, and `canBackGround flags`. Now make the PowerPC project, which produces a native code application. This completes the second stage, resulting in a fat binary application called SwitchBank. This one file runs on both 680x0 Macs and Power Macs.

Hazard

"Danger, Will Robinson!" says the robot. If you name the 680x0 file SwitchBank, and then try to create an PowerPC output file called SwitchBank, you can potentially confound the CodeWarrior IDE into a crash, or at least have it complain that the output file is busy. That's because it attempts to read resources from a file with the very same name as one to which you're trying to write PowerPC code.

To recap, by adding the 680x0 version of the application to the PowerPC project file, you fool CodeWarrior into automatically copying all of its resources—including the 680x0 CODE resources—into the PowerPC application at the completion of the second stage. You can confirm this by examining SwitchBank with ResEdit and seeing both cfrg and CODE resources. Although you could copy these resources using either ResEdit or Rez, the technique just described does the job using the two compilation stages you have to do anyway to make the 680x0 and PowerPC binaries.

There's one other thing you can do to SwitchBank so that it conserves memory on a Power Mac. CODE resources have an attribute bit set called Preload that makes the Resource Manager load them into memory automatically, whether they're used or not. You can fix this waste of memory with ResEdit. Launch ResEdit, and open the SwitchBank application. Open the CODE resource, and select all CODE segments but CODE 0. (For SwitchBank, there's only CODE segment 1.) Choose Get Resource Info from the Resource menu, or type Command-I. A Get Info box appears. Under the Attributes section, uncheck the Preload checkbox (see Figure 5.13). Save the file and quit ResEdit.

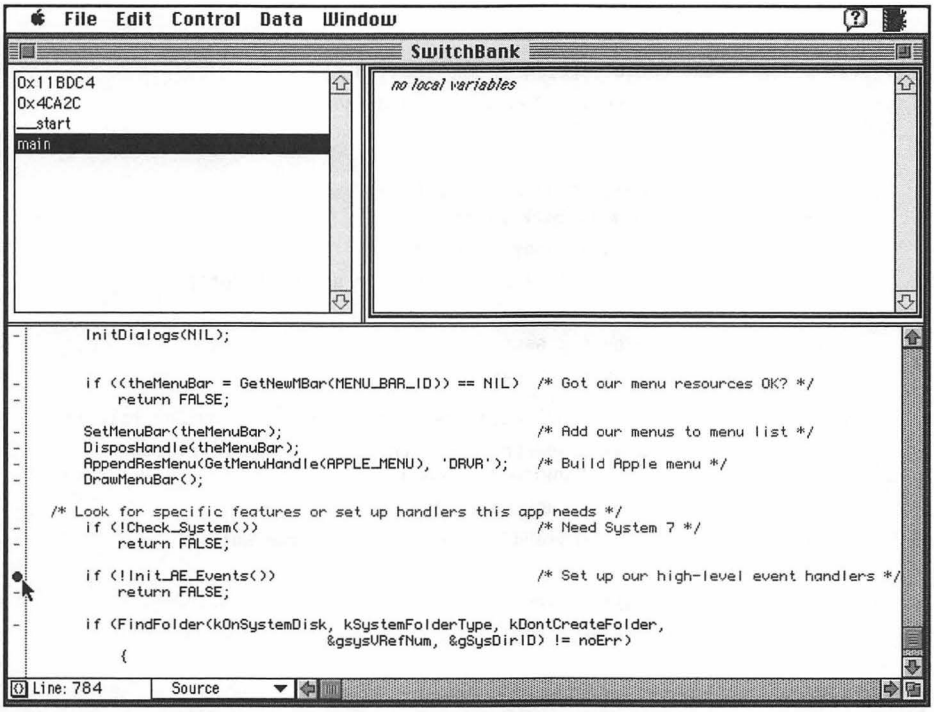


Figure 5.13 Changing the Preload attribute on a CODE resource.

Important

The CodeWarrior application supports a number of high-level Apple Events, including the four core Apple Events. The Metrowerks *CodeWarrior IDE User's Guide* describes the suite of Apple Events events that CodeWarrior IDE supports. These events let you create projects, adjust some of the preference settings of both project and output files, add or remove files from a project, compile files, and specify an output file. This capability enables you to automate portions of the development cycle, which is valuable for large or complex projects. Here's a sample AppleScript that generates a fat binary out of the SwitchBank code, using the settings in the SwitchBank project files to combine the two sets of code. Because you clear out the binaries, the PPC version of the project will automatically incorporate the latest 680x0 code present in SwitchBank.rsrc.

continues

continues

```
(* 1st stage - Make 680x0 version of application *)
tell application "YourHardDisk:CodeWarrior 8:MetroWerks
CodeWarrior:CodeWarrior IDE 1.4"
    activate
    open file "YourHardDisk:CodeWarrior 8:Code Examples
       ➥PPC:SwitchBank:SwitchBank.μ.68K"
    (* Project file should already have settings *)
    (* such as output file name and  its creator and type set*)
    Remove Binaries
    make project "SwitchBank.μ.68K"
    close project "SwitchBank.μ.68K"
(* 2nd stage - Make PPC version, using resources from 680x0 output file *)
   open file "YourHardDisk:CodeWarrior 8:Metrowerks CodeWarrior:Code
       ➥Examples PPC:SwitchBank:SwitchBank.μ.PPC"
   (* Project file should already have settings *)
   (* such as output file name and  its creator and type set*)
   Remove Binaries
   make project "SwitchBank.μ.PPC"
   close project "SwitchBank.μ.PPC"
   quit
end tell
```

Important

This is just a basic script, with the pathnames to the CodeWarrior IDE and project files hard-wired in. You'll have to edit these pathnames for this script to work on your system.

Handling a Code Fragment

Thus far you have seen how to supply the Mixed Mode Manager the information it needs to handle an instruction set switch when your custom function is called. Now it's time to go for an excursion into the Power Mac's basement, to get a glimpse of a code fragment close up. This brings you to the next utility, FlipDepth. Like SwitchBank, FlipDepth was a utility extension that I wrote to make my life easier. My work and interests are often at odds on a Mac's screen. The reason is that I make my living writing, with an occasional bit of code writing thrown in. In these situations, I need the utmost in scrolling speed when I examine a lengthy chunk of text or code listing. The

easiest fix, which costs you nothing, is to set the Mac's screen to black-and-white mode. This makes text scrolling very fast, because at this 1-bit pixel depth the Macintosh doesn't have as much data to pump to the screen as it does with color data. A color screen requires more bits per pixel, which means more data must be moved, and thus results in a slower scrolling process.

This wouldn't be an issue except that what I usually write about is heavy-duty graphics applications—the stuff that uses buckets of 24-bit pixels. So I was constantly clicking at the Monitors Control Panel, switching the Mac's screen depth from black-and-white to 24-bit color mode and back, depending upon what I was doing. If I could reduce the means of changing the screen depth to just a keystroke or two, it would make the job just a little easier. The real challenge is how to do this, of course.

Interlude: The Anatomy of a Trap

In Chapter 4, you learned that the 680x0 Mac's Toolbox routines are accessed via a dispatch table. Because much of the Power Mac's Toolbox is still 680x0 code, this remains true, although portions of the underlying mechanism that accomplishes this have changed.

Background Info

The Power Mac's 68LC040 emulator is made up of two components: a dispatch table (not to be confused with the 680x0 Mac's dispatch table) and a PowerPC code block. This dispatch table has an array of 64 K pairs of PowerPC instructions. The entries in the dispatch table correspond to 680x0 instructions. The main loop of the emulator fetches a 680x0 instruction word and uses it as a 16-bit unsigned index into the dispatch table. For simple 680x0 instructions, the first PowerPC instruction handles the operation, and the second instruction jumps back to the emulator loop. For complex 680x0 instructions, the first PowerPC instruction starts the emulation process, and the second instruction is a PC-relative branch into the code block. At this entry point are the PowerPC instructions that implement the 680x0 instruction. The emulator dispatch table also has entries for some of the A trap words, which point to native Toolbox routines. All A traps get routed through the standard 680x0 Mac dispatch

continues

continued

table, which exists on the Power Mac for compatibility. Execution either proceeds into the 680x0 emulator, or jumps to the emulator's dispatch table, and then to PowerPC code.

With System 7.5.2, a dynamic recompiling (DR) emulator boosts the execution speed of 680x0 instructions. The DR emulator is an add-on to the existing emulator, and uses the 680x0 dispatch table. Because of this, existing extensions and control panels that modify the trap tables still function.

The Mac Toolbox itself provides the means for you to reroute a Toolbox routine call to custom functions. Four routines, `GetOSTrapAddress()`, `SetOSTrapAddress()`, `GetToolTrapAddress()`, and `SetToolTrapAddress()`, provide a high-level interface that lets you change a 680x0 dispatch table entry, either for an OS call or Toolbox call. They accomplish this no matter how the run-time architecture establishes the connection between the Trap word and Toolbox routine's code. `GetOSTrapAddress()` and `GetToolTrapAddress()` accept a Toolbox trap word, and obtain from their respective dispatch table an address that points to the requested routine. (Recall in Chapter 4 that Trap dispatcher has two dispatch tables: one for OS services and the other for low-level routines.) `SetOSTrapAddress()` and `SetToolTrapAddress()` accept an address to your custom function and the Toolbox trap word. Both change the dispatch table's entry for this trap to point to your custom function instead.

Background Info

In the "classic" Mac era, the routines `GetTrapAddress()` and `SetTrapAddress()` patched the entries in one unified dispatch table. Because of the number of new services introduced with the Mac Plus, such as the Hierarchal File System and SCSI Manager, the dispatch table was divided into OS and low-level services. The routines `NGetTrapAddress()` and `NSetTrapAddress()` provided the means by which you modified these tables. Both routines required a third argument, `TrapType`, that specified the dispatch table to operate on. The initial version of FlipDepth used these routines, and they are still supported for compatibility. `GetOSTrapAddress()`, `SetOSTrapAddress()`, `GetToolTrapAddress()`, and `SetToolTrapAddress()` are the latest implementations of Apple's extension mechanism. They simplify extension writing by clearly stating what type of service they modify.

A bit of nomenclature here: These custom functions you write to modify a trap's behavior are called *patch code* because the term "patch" refers to fixing a hole in a wall by adding a little material, or fixing a software bug by adding a little code. Apple's system patches, which fix bugs or add enhancements, modify the dispatch table the same way to install additional code.

Now when an application calls the modified Toolbox routine, your function is called instead. If each Extension file's patch code does its job correctly, a call to a Toolbox routine can be reliably daisy-chained through several separate code enhancements before the actual Toolbox routine is invoked. Your function handles such a call one of two ways. The pseudo code for the first method looks like this:

```
My_Trap_Enhancement()
{
   Do_My_Stuff();
   Original_Trap_Routine();
} // end
```

This is called a *head patch*, because the function does its job first, then calls the Toolbox routine itself. Bear in mind that the pseudo code implies that the trap routine returns control to this function, when in reality it doesn't. Typically, `Do_My_Stuff()` performs its task, then jumps to `Original_Trap_Routine()`, never to return. You'll see an example of this shortly.

The second method uses this pseudo code:

```
My_Trap_Enhancement()
{
   result = Original_Trap_Routine();
   if (result == WHAT_WE_WANT)
         Do_My_Stuff();
} // end
```

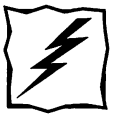
This is called a *tail patch*, because you call the Toolbox routine first, then perhaps act on a result returned by the routine. For example, you might call `MenuSelect()` and examine what it returns in order to act on a specific menu selection. Or, you might ignore what `Original_Trap_Routine()` does and instead perform a task based on the frequency that `Original_Trap_Routine()` gets called. Unlike the head patch, control does return to your function when the routine completes. On the 680x0 Mac, tail patches are considered evil because the return to your patch code can interfere with some of Apple's code patches

that work by examining the return address on the stack. For the Power Mac, the issue of how the patch is applied to a trap is moot, because its architecture is fundamentally different.

There are just a few more details you need to be aware of before you write a line of code. The Mac OS divides its memory into two sections: a system partition (or system zone) and an application partition (or application zone). Naturally, the Mac OS uses the system partition for its own use. The system partition contains the operating system's global variables (known as low-memory globals because they occupy some of the lowest physical addresses in RAM) and the system heap. The system heap is where drivers, patch code, and other resources hang out. Resources loaded here are typically shared by all applications. The application partition is where the Process Manager loads and launches applications. This section of memory is in constant flux as applications load and unload.

Your patch code has an important requirement: It can't move in memory. If it moves, even by accident, the pointer in the dispatch table (or in another Extension) winds up pointing at random data in memory, rather than at code. Thus a call to the patched Toolbox routine becomes a jump to nowhere, and the Mac crashes. You can lock the code in memory to prevent this, but you need to avoid creating an immoveable memory block that fragments the application partition. Thus, the system heap is an ideal place for the code.

Finally, there's the issue of accessing the global variables your patch code uses. At the very least, you need one global variable that stores the address you got from `GetOSTrapAddress()` or `GetToolTrapAddress()`, so that you can call the original routine. For the 680x0 Mac run-time architecture, this is a tricky matter. As you recall from the last chapter, register A5 points to an application's globals and jump table. When an application calls the patched Toolbox routine and your patch code executes, you have an immediate conflict of interest. Because the code is located somewhere else in memory, register A5 doesn't point to your globals. If you mess with A5 to correct this, there's the very real danger that you can mangle the application's A5 world. The application then loses track of its global variables and function references, which means certain death.



Important

Some more nomenclature: patch code typically belongs to a group of objects known as *stand-alone code*. Stand-alone code resources encapsulate pure machine code. These resources are loaded from a file into memory and executed directly. This is quite different from how an application loads. For an application, the Process Manager uses the file's CODE 0 resource to build an A5 world for it. Then the Process Manager jumps to another code resource that has the application's `main()` function. Because stand-alone code is executed without the benefit of any set up by the Process Manager, the value in A5 is meaningless. Also, stand-alone code has to be practically self-contained, because it can't rely on other resources being available, other than those supplied by the operating system.

Typical resources that include stand-alone code are drivers (DRVRs), custom window handlers (WDEFs), custom menu handlers (MDEFs), Control Panel code (cdev), and Extension code (INIT). Remember that last type, because you'll be returning to it shortly.

Fortunately, there's an easy fix, that was first pioneered by Symantec's THINK C, and is used by Metrowerks CodeWarrior. When you create a stand-alone code resource with the CodeWarrior IDE, it assumes that such code might be running concurrently inside of an application. The code it generates has all the references to global variables and to functions made with respect to register A4, rather than A5. When your code is called, all you need to do is call some glue code provided in a header file that sets up A4 to point to your code (and thereby our globals) for you.

The Power Mac's new run-time architecture simplifies how you handle globals. Because each code fragment has a separate data space, and a TOC that points to objects within it, ready access to global data is built in. To locate a certain global, you first find the code fragment you want by asking for it by name, and then asking for the global itself by name. You'll appreciate this more when you look at the actual code.

There is one problem to avoid when you patch a trap on a Power Mac. You want to avoid creating a performance hit with your patch code. Let's see why. Certain Toolbox routines in the Mac OS get called often by other Toolbox routines. (For example, `NewWindow()` calls QuickDraw routines to create a window on the screen, as does `DrawMenu()` in order to draw a menu.) Because the Power Mac's Toolbox is an amalgam of 680x0 and PowerPC code, these routines might get called by a 680x0 Toolbox routine one time, and then by a PowerPC Toolbox routine the next. A problem arises if this heavily called routine was only written in PowerPC code. The overhead of the Mixed Mode Manager performing the instruction set context switch for any 680x0 routine calling this particular routine becomes considerable. For small Toolbox routines, the context switch overhead becomes large enough to seriously degrade performance. Apple's solution was to implement these critical routines as "fat traps." That is, the routine is written in both 680x0 and PowerPC code. Regardless of what routine calls the fat trap, no context switch is required, and so the performance hit is minimized. The point here is that on the Power Mac, to avoid degrading performance of critical routines, you have to write a fat trap. This is very convenient, because it allows you to compare how to do a patch for both system architectures. However, be aware that not all traps have to be fat. For example, a heavily called routine that does a lot of processing would probably be better off patched only with PowerPC code, where the performance boost of native execution readily compensates for the overhead of the Mixed Mode Manager switch. A rough rule of thumb is that the overhead of the Mixed Mode Manager context switch takes approximately fifty 680x0 instruction equivalents or five hundred PowerPC instructions. If your patch function is roughly larger than fifty 680x0 instructions, then it's a candidate for being written as native code.

Writing a Fat Trap

With all of this information in hand, let's go write FlipDepth. Open the FlipDepth.µ.PPC file in the Code Examples PPC:FlipDepth:Projects folder. Or, start a new project in the CodeWarrior IDE and select ~Min MacOS PPC C/C++.µ from the Project Stationary popup menu. Use the editor to open FlipDepth.c in the FlipDepth:Sources folder:

```
/*
   Portions © 1994 Rock Ridge Enterprises.
   All Rights Reserved.
```

```
*/

/*
   This tells MixedMode.h that you want _real_
versions of    the various RoutineDescriptor
 functions and not dummy stubs.
*/
#define USESROUTINEDESCRIPTORS GENERATINGCFM

/*
   This #define is for testing only. Without it,
   only the    68K version of your patch is called.
*/
#undef DO_PPC_CODE_ONLY
//#define DO_PPC_CODE_ONLY

#pragma once on

#include <Memory.h>
#include <Gestalt.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <TextEdit.h>
#include <Files.h>
#include <Devices.h>
#include <Resources.h>
#include <Errors.h>
#include <Traps.h>
#include <LowMem.h>
#include <Events.h>
#include <Palettes.h>
#include <MixedMode.h>
#include <ConditionalMacros.h>
#include <CodeFragments.h>

#ifndef powerc
  #include <A4Stuff.h>
  #include <SetUpA4.h>
#endif

#define FALSE      false
#define TRUE      true
#define NIL        0L
```

```
/*
   Some low memory globals.
   You'd rather not use these,  but they're
   necessary because you'll be operating
 in a trap that doesn't move memory.
*/
#define lowMemKeyStroke    (*(KeyMap *) KeyMapLM)[0]
#define lowMemKeyModifiers (*(KeyMap *) KeyMapLM)[1]

/* Some constants that define the bits
   you'll see in KeyMap */
#define SHIFT_KEY      1L
#define CAPS_LOCK      2L
#define OPTION_KEY    4L
#define CONTROL_KEY    8L
#define COMMAND_KEY    0x8000L

#define KEY_COMBO      SHIFT_KEY + COMMAND_KEY
#define T_KEYCODE      0x0200L
#define BLACK_WHITE    128
   /* First video mode ID in sResource list */

#define FALSE      false
#define TRUE      true
#define NIL        0L

#define kOldSystemErr    10000
#define kMinSystemVersion  (0x0605)
```

Here's your usual complement of header files, plus definitions for the address of a low memory global and some constants. The header files you see here define information required by your job-specific code that controls the screen depth. One important thing to note is that you set USESROUTINEDESCRIPTORS to GENERATINGCFM (true) immediately before you include any header files. (Actually, you need this to happen before the MixedMode.h header file is used.) This way you signal the CodeWarrior IDE's 680x0 compiler that you are serious about supporting two processor instruction sets. The compiler then uses the universal header file's UPP-based descriptions for any Toolbox calls. A 680x0-specific compilation (`#ifndef powerc`), requires the SetUpA4.h and A4Stuff.h header files for the production of stand-alone code. Because you are using out-of-the-ordinary routines and settings here,

you may want to avoid use of the precompiled header files, such as MacHeaders.h, which ultimately calls MacHeaders68K.h or MacHeadersPPC.h (depending upon the target processor). If you really need the performance of the precompiled headers, edit and compile the MacHeaders68K.c or MacHeadersPPC.c files with the appropriate options set. Onward with your source code:

```
/*============================*/
#define kPPCRezType      'PPC '
#define kPPCRezID        300

/*==========================
   The 68k code goes in a normal INIT resource.
   Be sure this is set to "system heap/locked".
============================*/
#define kInitRezType     'INIT'
#define kInitRezID       128

/*================
   This is the name of the ppc fragment; for debugging only.
====================*/
#define kInitName     "\pEricsInit"

/*==========================
   To save some screen space, you'll use "UPP"
   instead of "UniversalProcPtr"
============================*/
typedef UniversalProcPtr   UPP;
```

Here are some more definitions, but now you are describing the characteristics of your generated code. Notice that you are declaring a resource type and ID number for a PowerPC code fragment here. What gives, when code fragments don't live in a file's resource fork? There are ways to access a code fragment from a file's data fork, but occasionally it's easier to load it from a resource. In taking this tack, you must make the PowerPC code fragment resemble a stand-alone code resource, so that the Mac OS treats it like one. This is accomplished by copying the PowerPC code fragment from a file's data fork into its resource fork, and then assigning the newly minted resource a type and ID number. The resource type doesn't have to be 'INIT', because you'll use a 680x0 INIT resource to actually install the PowerPC resource. Instead, define the resource as type 'PPC' so you can recognize it as PowerPC code. Now it's time to define your function prototypes:

```
/*============================
   PostEvent Information
============================ */
enum
{
   kPostEventInfo = kRegisterBased
      ¦ RESULT_SIZE(SIZE_CODE(sizeof(OSErr)))
      ¦ REGISTER_RESULT_LOCATION(kRegisterD0)
      ¦ REGISTER_ROUTINE_PARAMETER(1, kRegisterA0,
                        SIZE_CODE(sizeof(short)))
      ¦ REGISTER_ROUTINE_PARAMETER(2, kRegisterD0,
                        SIZE_CODE(sizeof(long)))
};

typedef pascal OSErr ( *PostEventFuncPtr ) ( short eventNum,
                              long eventMsg );
#define kPostEventFuncName "\pMyPostEventPPC"

/* Note separate functions */
short MyPostEvent68k( short eventNum, long eventMsg );
OSErr MyPostEventPPC( short eventNum, long eventMsg );

/*============================
   GetMouse Information
============================ */
enum
{
   kGetMouseInfo = kPascalStackBased
            ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Point)))
};

typedef pascal void ( *GetMouseFuncPtr ) ( Point *mouseLoc );
#define kGetMouseFuncName "\pMyGetMouse"

 /* Only one function required */
void MyGetMouse ( Point *mouseLoc );

 /*============================
  An original trap is called differently from PowerPC
  code than from 68k code because CallUniversalProc() and
  CallOSTrapUniversalProc isn't implemented for 68k code.
============================*/
#ifdef powerc
```

```
  #define CallPostEvent(eventNum, eventMsg) \
  CallOSTrapUniversalProc( gGlobalsPtr->gOrigPostEvent,
                kPostEventInfo, eventNum, eventMsg )
  #define CallGetMouse(mouseLoc) \
  CallUniversalProc( gGlobalsPtr->gOrigGetMouse, kGetMouseInfo,
              mouseLoc )
#else
  #define CallGetMouse(mouseLoc)
                    (*(GetMouseFuncPtr)
                    gGlobalsPtr->gOrigGetMouse)
                    ( mouseLoc );
#endif


/*==========================
  ShowInitIcon() defintions
========================== */
enum
{
  kShowInitIconInfo = kPascalStackBased
    ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
    ¦ STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(Boolean)))
};

// Function to display startup icon

typedef  pascal void (*ShowInitProcPtr) \
      (short iconFamilyID, Boolean advance);

#if powerc
typedef UPP ShowInitIconProcUPP;

#define CallShowInitIconProc(userRoutine,
                             iconFamilyID, advance)  \
  CallUniversalProc( (UPP)(userRoutine), kShowInitIconInfo,
              (iconFamilyID), (advance) )
#else
typedef ShowInitProcPtr ShowInitIconProcUPP;

#define  CallShowInitIconProc(userRoutine,
                              iconFamilyID, advance)  \
              (*(userRoutine))((iconFamilyID), (advance))
#endif
```

```
// Here you supply in-line definitions for NewRoutineDescriptor()
// and NewFatRoutineDescriptor() for a finicky linker
 #ifdef NewRoutineDescriptor
  #undef NewRoutineDescriptor
#endif
#ifdef NewFatRoutineDescriptor
  #undef NewFatRoutineDescriptor
#endif

extern pascal UPP NewFatRoutineDescriptor(ProcPtr theM68kProc,
      ProcPtr thePowerPCProc, ProcInfoType theProcInfo)
 TWOWORDINLINE(0x7002, 0xAA59);

extern pascal UPP NewRoutineDescriptor(ProcPtr theProc,
        ProcInfoType theProcInfo, ISAType theISA)
TWOWORDINLINE (0x7000, 0xAA59);

/* Custom function to place your patch */
/*  code in the system heap */
Handle Get1ResourceSys( OSType rezType, short rezID );

/* Functions that change screen depth. */
/* Works one both platforms */
void Change_Depth(long newDepth);
long Fetch_Depth(void);
```

These are the definitions for the Toolbox traps `PostEvent()`, `GetMouse()`, and `ShowInitProcPtr()`. `ShowInitProcPtr()` points to `ShowInitIcon()`, a custom function that displays the Extension file's icon on-screen during the boot process. The icon displayed by `ShowInitIcon()` indicates whether the patch code was installed properly or not. You examine `ShowInitIcon()` in more detail later. You also define the inline 680x0 macros for `NewRoutineDescriptor()` and `NewFatRoutineDescriptor()` so that these routines can be merged into your 680x0 initialization code.

Notice the scary macros `CallPostEvent()`, `CallGetMouse()`, and `CallShowInitIconProc()` define access to `PostEvent()`, `GetMouse()`, and `ShowInitIcon()`, respectively. They reduce using these routines to that of easy-to-read function calls. Otherwise, you would have to rely on extremely cryptic C code to do the job. Better still, the macros define a common way to call these routines, whether from 680x0 code or PowerPC code. On the PowerPC side of the fence, the macros handle the routine call by passing a

UPP and a routine descriptor to `CallOSTrapUniversalProc()` for an OS trap, or to `CallUniversalProc()` for either a Toolbox trap or function call. The only difference between these two routines is that `CallOSTrapUniversalProc()` preserves some additional 680x0 registers for register-based OS traps. For the 680x0 side, the macros just pass a pointers to the routines. Both techniques rely on addresses stored in a global block of memory. The macro `CallShowInitIconProc()` performs similar duties for the `ShowInitIcon()` display function, which is located in a separate resource. Note that for the 680x0 code, you don't declare a macro for `PostEvent()`. That's because a separate function is necessary to extract the values from this register-based OS routine.

Let's backtrack here a moment to explain why you are patching these two routines, `PostEvent()` and `GetMouse()`. I wanted to patch a trap that handled events, so that I could monitor the event stream for keystrokes. This way I could watch for the magic key combination that tells me the user wants to change the screen depth. `PostEvent()` is a Toolbox routine used by the Event Manager to place events in the event queue. It has two advantages: First, because it's actually responsible for creating the event stream, it's the perfect routine to monitor for keyboard events. Second, `PostEvent()` gets called frequently, so you can respond quickly to the user.

However, `PostEvent()` does have a down side. I've mentioned that it's an Operating System routine (or trap). Operating System routines typically perform low-level functions such as file I/O, network I/O, and memory management. In the Mac's early days, such routines were register-based. That is, the calling function passes information to the Operating System routine by placing the values in certain processor registers. This "calling" arrangement means that you are going to have to write assembly language to examine any values passed into or returned from this type of routine. The other problem is that `PostEvent()` doesn't move memory. Put another way, the routine's memory demands are fixed, so it's not going to force the Memory Manager to purge memory, or relocate data items whenever it's called. Lots of Toolbox routines and applications count on `PostEvent()` and certain other low-level OS routines being well-behaved about memory this way. Whatever the patch does, it has to be very simple lest you unexpectedly jar the location of objects in memory and cause a system crash. The safest thing to do is have the patched `PostEvent()` routine detect the right combination of key presses, and set a global flag. You'll use this flag to signal another patched routine to actually change the screen depth.

The function that handles the depth change should be patched into a Toolbox routine that doesn't have such strict memory requirements. That's because every application will be redrawing its chunk of the screen after the depth change, and this sort of thing definitely affects memory usage. Also, like `PostEvent()`, this routine should be called frequently for a fast response time. The routine `GetMouse()` fits these requirements.

`GetMouse()` is a stack-based Toolbox trap. That is, arguments are passed to this type of routine by pushing them on the stack. The result is typically returned on the stack, but there are exceptions. Ironically, `GetMouse()` is one of these exceptions, because it returns a result via a pointer you passed to the routine.

Important

Until now, I've used the term Toolbox loosely to mean any and all routines that implement services defined by the Mac API. For the moment, you'll have to make the distinction between Toolbox and OS traps. This is important because, as mentioned previously, the 680x0 Trap Dispatcher maintains two different dispatch tables: one for Toolbox traps and one for OS traps.

Keep in mind the discussion of these routines' memory behavior is based on the Mac's 680x0 architecture. However, because some of the Toolbox is emulated 680x0 code, you can assume similar behavior on a Power Mac. This will change over time as more of the Toolbox is rewritten as native code. Also, if you want the Extension to operate on the installed base of 680x0-based Macs, you need to follow the guidelines described previously.

Background Info

Historically, the OS traps were designed to be register-based because it was expected that these low-level routines would only be accessed by system programmers writing in assembly language. Toolbox traps, on the other hand, were made stack-based to make them easy to use. This was because application programmers would use these high-level routines in their applications.

> Nowadays, the distinction between the two trap types has blurred, because most compilers provide high-level access to OS traps using glue code. The definitions blur even further with the Power Macs, because all the routines pass their arguments through the PowerPC processor's registers, as described in Chapter 4.

Back to your code. These macros build routine descriptors that describe the makeup of the traps patched to the Mixed Mode Manager. Remember that your patch code ultimately calls the original trap, so you have to hand a routine descriptor to the Mixed Mode Manager so it can field an instruction set switch when one is required. You declare (no surprise) `PostEvent()` as a register-based routine and `GetMouse()` as a stack-based Pascal routine. `ShowInitIcon()` is defined as a stack-based Pascal procedure, a requirement brought about by the structure of a stand-alone code resource. You also define function prototypes for the routines and your patch code here. Observe that the `PostEvent()` patch code has both a PowerPC function and a 680x0 function. That's because for the 680x0 version, you have to do some processing in assembly language to retrieve `PostEvent()`'s arguments from the 680x0 processor registers. As you'll see, such gymnastics are unnecessary for the PowerPC version of the patch. Thus, the two different versions of the same patch.

Because you place the patch code in the system heap, you declare a custom function `Get1ResourceSys()` for this purpose. You also declare the screen control functions, `Get_Depth()` and `Change_Depth()`, here. You don't need to set up routine descriptors for these functions because they are called locally inside the patch code. Finally, you declare two function name strings, `kPostEventFuncName` and `kGetMouseFuncName`. The Code Fragment Manager uses these strings to locate your native patch functions.

```
/*==========================
   This structure is shared between the PowerPC
   version of the code and the 68K version.

   Both the PowerPC code and the 68k code have a single
   global variable, "gGlobalsPtr". They point to the
   same area of memory.
==========================*/
```

```
#ifdef powerc
   #pragma options align=mac68k
#endif

/*
   Note: do not move these fields around!
   The assembly code in PostEvent68kStub()
   depends on their locations. It must be
   compiled with the 68K packing conventions
*/

typedef struct
{
UPP      gOrigPostEvent;
// Address of original PostEvent trap
UPP      gOrigGetMouse;
// Address of original GetMouse trap
SysEnvRec  gSystemInfo;
// Holds info on system config
Boolean    gRequestFlag;
// Signals screen depth change
GDHandle    gOurGDevice;
// The GDevice of the screen
short      gDevRefNum;
// Driver num. for video board's slot
long      gOldScreenDepth;
// Mode num. for color screen setting
} MyInitGlobals;

#ifdef powerc
   #pragma options align=reset
#endif

/*==========================
   Global Variables

    -- Each side of the code maintains its own
      pointer to the same block of memory.

    -- You reference the globals ptr by name, so
      these two must be changed together.
==========================*/
MyInitGlobals              *gGlobalsPtr;
#define kGlobalsSymName    "\pgGlobalsPtr"
```

Here's your globals block, called `MyInitGlobals`. Note that you use the `#pragma options align=mac68k` to force word-alignment on the data structures so `MyInitGlobals` can be used on a 680x0 processor (or the 68LC040 emulator). The globals hold the original trap routine addresses (as UPPs, of course) and other sundry variables, such as the reference number to the device driver that controls the Mac's screen (`gDevRefNum`) and the logical device that manages it (`gOurGDevice`). Like your patch code, you also define a name string for the pointer to your globals. You'll pass this name to the Code Fragment Manager when you want to locate the globals block.

```
/*
@@@@@@@@@@@@@@@  68000 Exclusive Code   @@@@@@@@@@@@@@@
*/
#ifndef powerc

/*============================
   Prototypes for 68K code
============================*/
OSErr       DoInitForOldMacs( void );
OSErr       DoInitForPPCMacs( void );
OSErr       CreateFatDescriptorSys( void *mac68Code,
                        void *ppcCode,
                                    ProcInfoType procInfo,
                        UPP *result );
OSErr       PatchTrapsForPPCMac( ConnectionID connID );

void        PostEvent68kStub( void );
pascal void GetMouse68kStub ( Point *mouseLoc );
```

You now define some processor-specific functions. You'll use a 680x0-based INIT resource to set up and install your patch code, no matter what processor the Mac uses.

```
/*============================
   This is *always* the INIT's entry point. This is
   the only routine called by system software at startup.

   This requires that the INIT resource be set to
   System Heap/Locked.
============================*/
void main( void )
{
long            oldA4;
```

```
Handle          initH = nil;
    /* Handle to your own INIT resource */
OSErr            err = noErr;
long            ginfo;
ShowInitIconProcUPP  showCode;
Handle          showResource = NIL;

   /*****************************
      Global variable support
      Place proper value for A4 into hole in INIT resource.
   *****************************/
   oldA4 = SetCurrentA4();
      /* Get proper value of A4 into A4 */
   RememberA4();          /* save into self-modifying code */

   /*****************************
      Allocate your global variables
   *****************************/
   gGlobalsPtr = (MyInitGlobals*) NewPtrSysClear(sizeof
                                 (MyInitGlobals));
   if ( !gGlobalsPtr )
      {
      err = memFullErr;
      goto DONE;
      }

   /*****************************
      Get some basic system information
   *****************************/
   err = SysEnvirons( 1, &gGlobalsPtr->gSystemInfo );
   if ( err )
      goto DONE;

   /*****************************
      Check the system version
   *****************************/
   if ( gGlobalsPtr->gSystemInfo.systemVersion <
                              kMinSystemVersion )
      {
      err = kOldSystemErr;
      goto DONE;
      }
```

Here's the Extension's `main()` function, which loads and executes at boot time by the Mac OS. You first call the Metrowerks functions `SetCurrentA4()` and `RememberA4()`, which preserves register A4, then adjusts it to point at your code, and thus your globals. Next, you allocate a block of zeroed memory in the system heap, using `NewPtrSysClear()`. If you succeed at obtaining the memory, you then call `SysEnvirons()` to determine what operating system you are running under. If it's less than System 6.0.5, bail out, as you need the Gestalt Manager to tell you whether you are running on a Power Macintosh.

```
   /*******************************
      Get a handle to your own INIT resource
   *******************************/
   initH = Get1Resource( kInitRezType, kInitRezID );
   if ( !initH )
      {
      err = resNotFound;
      goto DONE;
      }

   /*******************************
      See whether you are running on a PowerPC
   *******************************/
   err = Gestalt( gestaltSysArchitecture, &ginfo );

   /*******************************
      Patch all the traps and get everything ready.
   *******************************/
   if ( err || (ginfo == gestalt68k) )
      err = DoInitForOldMacs();
   else
      err = DoInitForPPCMacs();

 DONE:
// Get ShowInitIcon() code
 showResource = Get1Resource('sdes', 128 );
 if (showResource != NIL)
   showCode = (ShowInitIconProcUPP) (*showResource);
       // Get pointer to resource header.
       // Don't need to lock down
       //   this resource because its
       //   attribute flags are marked
```

```
        //   as Locked and SysHeap

  // Something went wrong, clean up and display failure icon
  if ( err )
    {
    if ( gGlobalsPtr )
      DisposPtr( (Ptr)gGlobalsPtr );

    if (showResource != NIL)
      CallShowInitIconProc(showCode, (kInitRezID + 1),
                  TRUE ); // Display bad load icon
    }
  else
    // No initialization problems, do final setup and
    //display success icon
    {
    gGlobalsPtr->gRequestFlag = FALSE;
       // Clear request flag
    gGlobalsPtr->gOldScreenDepth = Fetch_Depth();
              // Get screen depth for later

  // Make sure the INIT code stays in memory when Extension
  // file closes. You do nothing for showResource
  // because you want it purged.

    DetachResource( initH );

    if (showResource != NIL)  // Display success icon
      CallShowInitIconProc(showCode, kInitRezID, TRUE);
    } /* end else */

  RestoreA4( oldA4 );    /* restore previous value of A4 */
} /* end main() */
```

Now, you set up a handle for your INIT resource. This resource contains the code you see here, the functions that patch the dispatch table, and your patch code. You load it into memory using `Get1Resource()`. Next, you use the Gestalt Manager to determine whether you are running on a Power Mac. If not, you call the function `DoInitForOldMacs()` to install the 680x0 patches. Otherwise, you call `DoInitForPPCMacs()` to handle the PowerPC patches.

Next, you load the code resource that contains `ShowInitIcon()`, to which `ShowInitProcPtr` points. You'll use this function to display the appropriate

installation icon, depending upon the results returned by `DoInitForOldMacs()` or `DoInitForPPCMacs()`. You access the function this way, rather than embedding it in your own code so that when you have completed FlipDepth's setup, the `ShowInitIcon()` code gets purged from the system heap. Because `ShowInitIcon()` is used only once, you can conserve memory by letting the Resource Manager dispose of the resource when it closes the Extension file. Ideally, you would organize FlipDepth so that `main()` and the initialization functions it calls are purged the same way, leaving only the actual patch code in the system heap. For the purposes of clarity and simplicity of design, however, I've kept the initialization code and patch code in one code segment.

If the patching operation fails, you clean up by releasing the memory allocated for your globals, and display the failure icon. In this case the Resource Manager eliminates the INIT code resource containing `main()` and your other setup functions when it closes the Extension file. If the patching process succeeds, you obtain the system's current screen depth for later use by your screen control functions, take steps to keep your code in the system heap, and display the success icon.

To ensure that the Resource Manager doesn't purge your INIT code from memory, you call `DetachResource()`. This Toolbox routine severs the logical link between the Resource Manager and this resource, so that the resource remains in memory when the Resource Manager closes the file. Because you don't detach the `ShowInitIcon()` resource, it's disposed for you automatically. Note that you check for a valid handle to `ShowInitIcon()`'s code resource before attempting to use it. Finally, restore register A4 and exit.

```
OSErr DoInitForOldMacs( void )
{
  /* patch the traps */

  gGlobalsPtr->gOrigPostEvent = GetOSTrapAddress( _PostEvent );
  SetOSTrapAddress( (UPP)PostEvent68kStub, _PostEvent );
  gGlobalsPtr->gOrigGetMouse = GetToolTrapAddress( _GetMouse );
  SetToolTrapAddress( (UPP)GetMouse68kStub, _GetMouse );

  return noErr;
} /* end DoInitForOldMacs() */
```

Here's where you modify the 680x0 Mac's dispatch table to point to your patch code, and it's pretty straightforward. You first obtain the original trap address from the appropriate dispatch table using either `GetOSTrapAddress()` or

GetToolTrapAddress(), and save it in your globals block, MyInitGlobals. Next, use SetOSTrapAddress() or SetToolTrapAddress() to replace the original address with a UPP (actually, a 680x0 procedure pointer) to your patch code. Now let's see how it's done for a Power Mac:

```
/*==========================
   DoInitForPPCMacs

   Initialization code for powerpc Macs.
==========================*/
OSErr DoInitForPPCMacs( void )
{
OSErr          err = noErr;
Handle         ppcCodeH = nil;
SymClass       theSymClass;
Ptr            theSymAddr;
ConnectionID   connID = kNoConnectionID;
Str255         errName;
Ptr            mainAddr;

   /*******************************
      Load the powerpc version of the code into
      memory. Because some of your trap patches may be
      called at interrupt time, don't use disk-based
      versions of the code.
   *******************************/
   ppcCodeH = Get1ResourceSys( kPPCRezType, kPPCRezID );
   if ( !ppcCodeH )
      return resNotFound;
   HLock( ppcCodeH );

   /*******************************
      Open a connection with the code fragment you just loaded
   *******************************/
   err = GetMemFragment( *ppcCodeH, GetHandleSize(ppcCodeH),
                  kInitName, kLoadNewCopy, &connID,
                  &mainAddr, errName );
   if ( err )
      {
      connID = kNoConnectionID;
      goto DONE;
      }
```

Because the container for your Code Fragment is a resource, you must first load it into memory with the Resource Manager. You do this using your custom function `Get1ResourceSys()`, which loads the fragment into the system heap and returns a handle, `ppcCodeH`, to it. `Get1ResourceSys()`, described later, sets memory accesses to the system partition and then calls the Resource Manager to load the resource into that partition. You lock this code in place using `HLock()`. Because your `PostEvent()` patch code might get called during an interrupt, it requires that the patch code remain in memory at all times, which is why you load the fragment into the system heap and lock it in place. Now you pass the code fragment's handle to `GetMemFragment()`, which prepares the fragment for execution. You use `GetMemFragment()` over other Code Fragment Manager routines because it operates on fragments in memory. The constant `kLoadNewCopy` has `GetMemFragment()` make a new copy of any of the fragment's writable data (like your globals), and `connID` returns an ID value that specifies a connection to this fragment. You could also use the constant `kLoadLib`. The connection ID is analogous to the file reference number that the File Manager routines use for file I/O. You supply this connection ID to other Code Fragment routines to obtain information on fragments, or the addresses of functions or global data within fragments. Now it's time to find those globals:

```
/*******************************
   find the global variable ptr that the powerpc
   code uses.
*******************************/
err = FindSymbol( connID, kGlobalsSymName, &theSymAddr,
          &theSymClass );
if ( err )
   goto DONE;

/*******************************
   Modify the powerpc global variable pointer to point
   to the area of memory you have already allocated.
*******************************/
*(MyInitGlobals **)theSymAddr = gGlobalsPtr;
err = PatchTrapsForPPCMac( connID );

/*******************************
   Cleanup
*******************************/
DONE:
```

```
    if ( err )
        {
    /* Close the code frag mgr connection if you got an error */
        if ( connID != kNoConnectionID )
            CloseConnection( &connID );

    /* ...and release the memory you allocated */
        if ( ppcCodeH )
            ReleaseResource( ppcCodeH );
        } /* end if */
    else
        {
    /* No error -> keep the ppc code around when file closes */
        DetachResource( ppcCodeH );
        } /* end else */

    return err;
} /* end DoInitForPPCMacs() */
```

You use the Code Fragment Manager routine `FindSymbol()` to locate the PowerPC version of your globals pointer, `gGlobalsPtr`. You pass it the connection ID obtained with `GetMemFragment()`, and the export name of your globals pointer in the string `kGlobalsSymName`. `FindSymbol()` returns the address of the pointer in `theSymAddr`. You then direct this pointer toward your globals block. Now that you can locate your globals, you call `PatchTrapsForPPCMac()` to patch the dispatch table. If all goes well, you call `DetachResource()` on the PowerPC resource to make the Resource Manager "forget" about the fragment and leave it in memory. If there is an error, you close the connection to the code fragment using `CloseConnection()`, and follow that with a call to `ReleaseResource()` to dispose of the code fragment.

Let's see how you patch traps on the PowerPC run-time architecture:

```
/*============================
   PatchTrapsForPPCMac
============================*/
OSErr PatchTrapsForPPCMac( ConnectionID connID )
{
Ptr             symAddr;
SymClass           symType;
OSErr            err = noErr;
UniversalProcPtr    upp = nil;
```

```
//    Fat Patch _PostEvent

  err = FindSymbol( connID, kPostEventFuncName, &symAddr,
              &symType );
  if ( err )
    return err;

  err = CreateFatDescriptorSys( PostEvent68kStub, symAddr,
                      kPostEventInfo, &upp );
  if ( err )
    return memFullErr;

  gGlobalsPtr->gOrigPostEvent = GetOSTrapAddress( _PostEvent );
  SetOSTrapAddress( (UPP)PostEvent68kStub, _PostEvent );

//    Fat Patch _GetMouse

  err = FindSymbol( connID, kGetMouseFuncName,
              &symAddr, &symType );
  if ( err )
    return err;

  err = CreateFatDescriptorSys( GetMouse68kStub, symAddr,
                      kGetMouseInfo, &upp );
  if ( err )
    return memFullErr;

  gGlobalsPtr->gOrigGetMouse = GetToolTrapAddress( _GetMouse );
  SetToolTrapAddress( (UPP)GetMouse68kStub, _GetMouse );

  return noErr;
} /* end PatchTrapsForPPCMac() */
```

`FindSymbol()` greatly simplifies matters here. You provide this routine with the name of your patch code functions, and it returns the entry points to them in the code fragment. Because the course of execution could be hopping from one instruction set to another, you next build a routine descriptor for these functions. `CreateFatDescriptorSys()` is a custom function that places the descriptor information in the system heap. You examine its code shortly. You call this function with the address of your 680x0 patch code, the address of your PowerPC patch code, and the routine descriptor information provided at the start of this file. `CreateFatDescriptorSys()` returns a UPP that points to

both the 680x0 patches and PowerPC patches. At this point, patching the Power Mac's dispatch table is nearly identical to how it's managed with the 680x0 dispatch table. The original trap address is copied from the proper dispatch table using either `GetOSTrapAddress()`, or `GetToolTrapAddress()`, and it's replaced with the UPP to your patch code by calling `SetOSTrapAddress()` or `SetTrapAddress()`. You could make this section of code more robust by performing the memory allocations (via `CreateFatDescriptorSys()`) and symbol locations in the `main()` function. This way, if there's a problem applying either of the patches, you have a chance to back out gracefully.

```
/*==========================
   CreateFatDescriptorSys

   Creates a fat routine descriptor in the system heap.
==========================*/
OSErr CreateFatDescriptorSys( void *mac68Code, void *ppcCode,
                    ProcInfoType procInfo, UPP *result )
{
THz   oldZone;
OSErr err = noErr;

   oldZone = GetZone();        /* Save current zone */
   SetZone( SystemZone() );    /* Get you in the system heap */

   #ifndef DO_PPC_CODE_ONLY
   *result = NewFatRoutineDescriptor( mac68Code, ppcCode,
                         procInfo );

   #else        /* debugging only */
   *result = NewRoutineDescriptor( ppcCode, procInfo,
                       kPowerPCISA );
   #endif

   SetZone( oldZone );

   return ( *result ? noErr : memFullErr );
} /* end CreateFatDescriptorSys() */
```

Here's that custom function that generates routine descriptors in the system heap. You begin by saving the current zone (or memory partition). This is done by first calling `GetZone()` to obtain a pointer to this zone, and saving it in `oldZone`. Then you change the zone that you operate in to the system zone. To

do this, you call `SystemZone()` to get a pointer to the system heap, and make it the active zone by passing this pointer to `SetZone()`. Now when you generate a new data structure, such as your fat descriptor, the memory gets drawn from the system heap. Then you call `NewFatRoutineDescriptor()`, which makes the UPP containing a fat descriptor. Once that's done, you restore the current zone by passing `oldZone` to `SetZone()`, and exit.

```
/*============================
   PostEvent68kStub
============================*/

asm void PostEvent68kStub( void )
{
// Reserve space on stack for "real" PostEvent address
   sub.l       #4, SP
// Save registers (not A0 & D0, though)
   movem.l     A1-A5/D1-D7, -(SP)

// Push A0 & D0 on stack for call to MyPostEvent68k below
// You must do this before SetUpA4 because it modifies registers
   move.l      D0, -(SP)      // push event message
   move.w      A0, -(SP)      // push event code

   jsr         SetUpA4        // give you global access

// Put address of "real" postevent in place reserved on stack
// Note that it is the first field in the gGlobals structure
   move.l      gGlobalsPtr, A0
   move.l      (A0), 54(SP)

// Call MyPostEvent68k
// Parameters are on the stack already
// D0.w returns with the new event code
   jsr         MyPostEvent68k

   move.w      D0, A0       // A0.w = event code
   add.l       #2, SP       // Clear old event code from stack
   move.l      (SP)+, D0  // Restore event message from stack

// restore registers
   movem.l     (SP)+, A1-A5/D1-D7
```

```
      // Jump directly to original PostEvent code
      // The address was placed on the stack in the above code
   rts
} /* end PostEvent68kStub() */

pascal void GetMouse68kStub( Point *mouseLoc )
{
long     oldA4;

   oldA4 = SetUpA4();
   MyGetMouse ( mouseLoc );
   RestoreA4( oldA4 );
} /* end GetMouse68kStub() */

#endif      /* 68K code */
```

These are the 680x0 code stubs for your patch code. These stubs minimally fix up register A4 to point to your globals before calling your patch code, and restore A4 when they exit. As you'll see in a moment, for OS traps the stub has a lot more work to do. `PostEvent68kStub()` is the entry point for the 680x0 `PostEvent()` patch code and is a head patch. You use CodeWarrior's built-in 68K assembler to write 680x0 assembly-language code that fetches the contents of register A0, which contains the event code (or type), and the contents of register D0, which holds the event's message. It's a nasty business, because you have to keep careful track of where things are on the stack. There are two things to be aware of with the CodeWarrior's built-in 68K assembler. First, you can't place assembly language instructions directly inline with C code, as you can with Symantec's THINK compiler. The assembly language code must be wrapped inside a function. This function is declared `asm`, as you can see in the code. Second, to comment assembly-language statements you use C++ style comments, where each comment is lead with a double-slash (//).

When `PostEvent68kStub()` gets called, you first save room on the stack where you'll stow the address of the original `PostEvent()` trap. Then you save most of the processor registers. Next, you retrieve `PostEvent()`'s arguments out of register A0 and D0 and push them onto the stack, for use in your patch function `MyPostEvent68K()`. Now you call `SetUpA4()` to fix up register A4 so you can get at your globals. This lets you obtain the pointer to your globals block, `gGlobalsPtr`. Once that's done, you fetch the address of the original `PostEvent()` from `gOrigPostEvent` and drop it on the stack. Because `gOrigPostEvent` starts the

globals block, you don't need an offset from the pointer to access it. You stuff this address into the location on the stack where you allocated room for it. Because of all of the items you have pushed onto the stack so far, this location is 54 bytes from the current stack top.

With all the preliminary setup done, you at last call `MyPostEvent68k()`, the patch code which processes the event. When it returns, you place the event code it returns back into A0. You then toss the original event code into the bit bucket (because `MyPostEvent68k()` might have changed it), move the original event message back into D0, and restore the registers. At the end of all this work, the address of the original `PostEvent()` has moved to the top of the stack, and so that routine gets called when your function exits.

Because `GetMouse()` is a stack-based routine, all `GetMouse68kStub()` does is set up access to your globals using `SetUpA4()` before calling the real patch code in `MyGetMouse()`. You restore A4 as the function exits.

```
/*
   @@@@@@@@@@@@@@@  Shared Code   @@@@@@@@@@@@@@@

   This code gets compiled into both 68k and powerpc object code.
   The 68k code gets called from 68k patches & code.
   The powerpc code gets called from powerpc patches & code.

   If these routines were very large, or called infrequently, you
   could just have a single version that is called by the
  "other" object code, but it's not worth the hassle
  & context switch.
*/
Handle Get1ResourceSys( OSType rezType, short rezID )
{
   THz          oldZone;
   Handle       h;

   oldZone = GetZone();
   SetZone( SystemZone() );
   h = Get1Resource( rezType, rezID );
   SetZone( oldZone );
   return h;
}

/* Your custom GetMouse function. You do your screen stuff here because _GetMouse
is allowed to move memory, and is called frequently.
```

```
*/

void MyGetMouse( Point *pt )
{
long  currentDepth;

   if ( gGlobalsPtr->gRequestFlag )    /* Event is for you ? */
      {
      gGlobalsPtr->gRequestFlag = FALSE; /* Clear flag */
      currentDepth = Fetch_Depth();
      if ((currentDepth == BLACK_WHITE) &&
         (currentDepth != gGlobalsPtr->gOldScreenDepth))
         Change_Depth(gGlobalsPtr->gOldScreenDepth);
      else
         Change_Depth(BLACK_WHITE);
      } /* end if */

   CallGetMouse( pt );      /* Hop to original GetMouse() */

} /* end ourGetMouse() */
```

Most of the functions here, with the exception of the `PostEvent()` patch code, are compiled for both processors. The resulting machine code goes into separate resources ('INIT' for 680x0 code and 'PPC ' for PowerPC code) to build the fat trap, with a fat descriptor pointing to the function entry points in each resource.

The function `Get1ResourceSys()` loads the specified resource into the system heap. `MyGetMouse()` is the patch code for the `GetMouse()` routine. When it's called, it checks to see whether `gRequestFlag` has been set. If so, it knows that the user requested a screen depth change. The function first clears this flag so that it won't respond again the next time the routine gets called. `MyGetMouse()` next has `Fetch_Depth()` determine the current screen depth. This is checked against the constant `BLACK_WHITE` and the mode of the screen depth that was saved when the Extension loaded. If the screen depth mode doesn't match `BLACK_WHITE`, then it calls `Change_Depth()` to set the video hardware to display the shallower pixel depth. If the screen depth matches `BLACK_WHITE`, it calls `Change_Depth()` to switch the video hardware back to the original display mode with a deeper pixel depth. The reason for the complicated `if` statement is to head off potential trouble if you start the Mac with its screen at the shallowest pixel depth, typically the black-and-white mode (hence the name of the constant). In this case, FlipDepth has no idea what other screen depths the

video hardware supports, so the code locks the screen into this mode. If it didn't, `Change_Depth()` would be called with a garbage value, which might result in an interesting, if unusable, screen. Once you have changed the screen depth, you call the original `GetMouse()` routine to finish the call.

```
#ifdef powerc

OSErr MyPostEventPPC( short eventNum, long eventMsg )
{
OSErr  result;

  if ( (eventNum == keyDown) || (eventNum == autoKey) )
    {
    if ( (lowMemKeyModifiers == KEY_COMBO) &&
      (lowMemKeyStroke == T_KEYCODE) )
      {
      eventNum = nullEvent;    /* Supress the event */
      gGlobalsPtr->gRequestFlag = TRUE;
      } /* end if KEY_COMBO && T_KEYCODE */
    } /* end if */

  result = CallPostEvent(eventNum, eventMsg);
  return result;
} /* end MyPostEventPPC() */

#else    // 68K code

/*
  Note:
  returns the (possibly modified) event code

  Don't modify the local variables eventNum & eventMsg
    -- they're used by the stub routine and modifying
    -- locals here can have a global effect
*/
short MyPostEvent68k( short eventNum, long eventMsg )
{
short  newEventCode = eventNum;

  if ( (eventNum == keyDown) || (eventNum == autoKey) )
    {
    if ( (lowMemKeyModifiers == KEY_COMBO) &&
      (lowMemKeyStroke == T_KEYCODE) )
```

```
        {
        newEventCode = nullEvent;  /* Supress the event */
        gGlobalsPtr->gRequestFlag = TRUE;
        } /* end if KEY_COMBO && T_KEYCODE */
      } /* end if */

  return newEventCode;
} /* end MyPostEvent68k() */

#endif
```

These functions are the 680x0 and PowerPC versions of the `PostEvent()` patch. Basically, they watch the event code (or its type) passed to the routine. Because you are looking for a special key-combination, the code ignores all events but key down and auto key events. If a keyboard event occurs, you examine a low memory global, `KeyMapLM`, to determine what keys were pressed. You would rather not use a low memory global because it introduces an absolute address in your code, but other routines that could do the job also happen to move memory.

Hazard

Apple will eventually phase out certain low memory globals, because they hamper moving the Mac OS to a preemptive multitasking operating system, particularly Copland. Therefore, the use of low memory globals is strongly discouraged. However, for the FlipDepth example you have two choices. The first is to perform a safe head-patch on `PostEvent()` that uses a low-memory global still supported on second-generation Power Macs to obtain the modifier keys. In short, FlipDepth, as implemented here, works reliably on machine architectures both now and for the immediate future. Or, the second choice is to avoid using the low memory global by performing a tail-patch on a Toolbox call such as `GetOSEvent()`, to capture both the event and the modifier keys. This might buy you trouble immediately if your tail patch interferes with Apple's patch software. When you're dealing with the Mac OS at this level, sometimes there are no easy choices.

`PostEvent()`'s flaw is that monitoring the keyboard this way isn't very portable. Specifically, the same key code can mean something entirely different on a French or Kanji keyboard. A better routine to patch was

> `PPostEvent()`, where you can peek into the event queue after the event is posted to obtain the status of the modifier keys. This works in favor of code portability, because the Event Manager OS processes the keystrokes so that these raw key codes map to the country-specific modifier keys.

If Command-Shift-T is pressed, it's a request to change the screen depth. You respond by clearing the event code to discard the event. If you don't do this, the keyboard event is forwarded to the application, which might respond in undesirable ways. Then, you set the global `gRequestFlag` and exit.

Before you could call the 680x0 version of this function (`MyPostEvent68k()`), you had to do some scary assembly code to position the arguments onto the stack where you could use them. This isn't the case for the PowerPC version. Even though `PostEvent()` is register-based, when `MyPostEventPPC()` is called, these values appear in the function's arguments, as if the routine were stack-based. This simplifies use of the OS trap routines immensely, thanks to the Mixed Mode Manager.

As a final note, when `MyPostEvent68k()` exits, it has to traverse more assembly code to clean up the stack, restore register A4, and jump to the original `PostEvent()`. The PowerPC version simply calls the `CallPostEvent()` macro and exits. Although it's possible to write a `PostEvent()` function using CodeWarrior's PowerPC assembler, there's no good reason to do so. As you've just seen, writing the patch code in C is adequate.

But I digress. Onward to the screen depth control software:

```
/*  Get the current screen depth. Also get the GDevice of main screen and its */
/* device number (to use the driver) */

long Fetch_Depth(void)
{
long     screenDepth;
     /* Current bit depth of your screen */
GDHandle thisGDevice;

   /* Get start ;0f GDevice list */
   thisGDevice = GetMainDevice();
     /* Get GDevice of main screen */
   gGlobalsPtr->gOurGDevice = thisGDevice;
   screenDepth = (**thisGDevice).gdMode;
```

```
   /* Get pixel's size */
/* Driver # */
   gGlobalsPtr->gDevRefNum = (**thisGDevice).gdRefNum;
   return screenDepth;

} /* end Fetch_Depth() */
```

`Fetch_Depth()`'s job is to find the Mac's main active screen. It uses a call to the routine `GetMainDevice()`, which fetches the `GDevice` for the main screen. A `GDevice` is a data structure used to maintain screen information for both the display hardware and the Mac OS. It stores information, such as the screen's size, current color palette, the device driver controlling the display hardware, and the screen's pixel depth. Next, you obtain the driver reference number and current screen mode from this `GDevice`, and place this data in the globals `gdRefNum` and `gdMode`. The mode value is the ID number of a special resource used to handle the screen.

```
void Change_Depth(long newDepth)
{
GrafPtr     oldPort;
Rect        ourGDRect;
RgnHandle   thisScreenBoundary;
GrafPtr     theBigPicture;
WindowPtr   theFrontWindow;

   HideCursor();
     /* Hide pointer because its depth will change */
   InitGDevice(gGlobalsPtr->gDevRefNum, newDepth,
      gGlobalsPtr->gOurGDevice);
/* At last you change the screen depth! */
   theFrontWindow = FrontWindow();
   ActivatePalette(theFrontWindow);
     /* Use active window's color palette */
   AllocCursor();      /* Draw cursor at new screen depth */
   ShowCursor();      /* Put it back on-screen */

/* The desktop's still a mess: redraw it */
   thisScreenBoundary = NewRgn(); /* Get region to hold screen */
   if (!MemError())               /* Trouble? */
      {                           /*No */
      ourGDRect = (**gGlobalsPtr->gOurGDevice).gdRect;
 /* Get gDevice boundary */
      RectRgn(thisScreenBoundary, &ourGDRect);
```

```
        GetPort(&oldPort);            /* Save current port */
        GetWMgrPort(&theBigPicture);  /* Get Desktop's port */
        SetPort(theBigPicture);       /* Make it the current port */
        DrawMenuBar();
        PaintOne(NIL, thisScreenBoundary); /* Paint background */
    /* Now the other windows */
        PaintBehind( *(WindowPeek *) WindowList,
                thisScreenBoundary);
        SetPort(oldPort);
        DisposeRgn(thisScreenBoundary);
        } /* end if !MemError() */
     else
        SysBeep(30); /* Couldn't make the region, complain */

} /* end Change_Depth() */
```

Last but not least, here's the function that changes the video hardware's pixel depth. The second line of code, where `InitGDevice()` is called, does the actual depth change. You pass this routine the device reference number so that it can communicate with the driver that controls the screen's video hardware, the new screen mode value, and the `GDevice` that manages the screen. `Fetch_Depth()` conveniently obtained the display's driver reference number and its associated `GDevice` that you now use in this `InitGDevice()` call. The rest of the code in this function basically cleans up the screen after the depth change.

Let's talk about those screen modes a bit more. The screen mode number derives from the ID numbers of special resources (called sResources, because they're Slot Manager resources) in a display board's firmware (either NuBus or PCI), or in firmware that manages the Mac's built-in video circuits. Each different pixel depth that the display supports has its own sResource ID number. These sResources contain information that describes the screen's characteristics to both the operating system and the device driver for a particular screen depth (say, 8 bits per pixel).

What's key here is that these sResources are handled a lot like actual resources, where the first available ID number begins at 128. The Macintosh API dictates that the first screen mode always have the shallowest pixel depth, and its mode sResource value must always be 128. If you call `InitGDevice()` with a mode value of 128, the screen turns switches to the shallowest pixel depth. How do you handle other screen depths? You punt on that issue, because there's no guarantee as to what pixel depth the next

sResource (ID = 129) supports. A display board might support 1-, 2-, 4-, and 8-bit color, so its sResource IDs would be 128, 129, 130, and 131, respectively. Another board might support 1-, 8-, 16-, and 32-bit screen modes, and its sResources would also be 128, 129, 130, 131. FlipDepth grabs the current screen mode (and thus its sResource ID number) when the Extension loads, and saves it in the globals block. You simply pass this value, and whatever screen depth it represents, to `InitGDevice()` whenever the user wants to leave the shallowest pixel-depth mode. While this all sounds complicated, the code shows that it's fairly simple. The big payoff is that this mechanism is hardware independent: this identical code works on Mac IIs, PowerBooks, Quadras, and Power Macs, including the second-generation Power Macs, which use PCI rather than NuBus display boards.

Important

Why don't I use the high-level routines `HasDepth()` and `SetDepth()`, which obtain a screen mode and set a screen's mode, respectively? I wrote this code long before these routines appeared on the scene. Also, the initial release of these routines was slightly buggy. As a fast hack for a screen utility, this code has served me well for many years, and is still hardware-independent. The second-generation Power Macs use an Expansion Manager, rather than a Slot Manager, to handle communications between the Mac OS and the display hardware. Nevertheless, the Expansion Manager still uses the `GDevice` data structure for housekeeping, which means the display mode value is still valid, even though its meaning has changed slightly. Originally, a screen mode value of 128 represented a black-and-white screen depth. Depending upon some Power Mac configurations, this is no longer true. For example, the shallowest pixel depth on ATI's PCI display board is 256 colors (8-bits).

The rest of this function handles repainting the screen after the depth changes. You start by hiding the cursor, and do the depth change. Next, you fix up the color palette so that it uses the color palette of the foreground application (which owns the front window) with a call to `ActivatePalette()`. Then you fix the cursor's pixel depth.

Redrawing the screen itself requires that you obtain a region that you use to map the desktop onto so that you can redraw the background pattern. You use `NewRegion()` to make this region structure. You plug into this region the boundaries defined by the screen's `GDevice`, using `RectRgn()`. The current drawing port is saved, and you use `GetWMgrPort()` to fetch the port that handles the entire desktop. You make this the current drawing port and call `DrawMenuBar()` to reconstruct the menus. `PaintOne()` is a Window Manager routine that, when called with a value of NIL for the window argument, knows that the "window" is the desktop and paints it with the background pattern. `PaintBehind()` then redraws all the windows in the region. At this point, the Mac's screen is rebuilt, so you clean up by restoring the port and releasing the memory used to make the region.

Building a Fat Trap

At last, you are ready to use this code to build your fat trap. Similar to what you did to make a fat binary of your SwitchBank program, you have two project files. The project FlipDepth.µ.PPC generates PowerPC patch code from FlipDepth.c, while FlipDepth.µ.68K generates the 680x0 half of the patch code. Because a 680x0 INIT resource handles the Extension's setup, you build the PowerPC version of the patch code first, and then integrate the results of this operation into the 680x0 version of the Extension file. You have already opened the project file FlipDepth.µ.PPC in the Projects folder, and its project window should already have InterfaceLib, MathLib, and MWCRuntime.Lib. Remove the last two files from the project, leaving only InterfaceLib.

Now it's time to adjust the project's preferences for your Extension file. You want to make a shared library in order to handle the code fragment as stand-alone code. If you don't, the linker will add some run-time code that prepares the code fragment for execution as an application when it loads. First, select Preferences from the CodeWarrior IDE's Edit menu and go to the PPC Linker panel. On the Entry Points section of this panel, clear all three text boxes, Initialization, Main, and Termination, of their default entry point names. To recap, as a stand-alone resource, you don't want any run-time code trying to do something with your code fragment as it loads, which is why you jettisoned MWCRuntime.Lib and cleared the fragment's entry points.

Now, onward to the other project settings. First, select the PPC Processor panel, and chose 68K alignment from the Struct Alignment's popup menu. Inside the PPC PEF panel, go to the Export Symbols popup menu item and select Use ".exp" file. Choose the Fragment Name item in this panel and type **`PPC `** (don't forget to add the space at the end of this resource type name!). Make sure that the item Share Data Section is checked. The next stop is the PPC Project panel. Click the Project Type's popup menu and choose Shared Library. The window's contents change, displaying items that modify the shared library's characteristics. Go to the Shared Library Info section. Type the name **`FlipDepth.lib`** for the library's name and leave the file's type and creator alone. Confirm that the file type is 'shlb.' Save all settings by clicking the OK button. Click the Make button in the Toolbar, or select Make from the Project menu, to compile the code and create the library. If all has gone well, you should have the files FlipDepth.lib and FlipDepth.µ.PPC.exp in your FlipDepth:Projects folder. If the files don't appear, recheck the settings in the PEF panel.

You'll recall in the PPC PEF panel that you selected an ".exp" file for handling symbol export, and a "FlipDepth.µ.PPC.exp" file was made. This file contains the symbols that you wish to export, or make public, to other code fragments. You would use this to, say, make public the entry points into a shared library you wrote, while hiding the functions that implement the library's services. By minimizing the number of public functions, you also reduce clutter in the code fragment's TOC. Now let's see how this is done. Use the editor to open the "FlipDepth.µ.PPC.exp" file.

In it, you'll see a number of function and global variable names: `Change_Depth`, `Fetch_Depth`, `Get1ResourceSys`, `MyPostEventPPC`, `MyGetMouse`, and `gGlobalsPtr`. Delete all of these names except `gGlobalsPtr`, `MyPostEventPPC`, and `MyGetMouse`. The next time you build FlipDepth, the PPC linker uses the information in the modified FlipDepth.µ.PPC.exp file to make only these three items (the globals data block, and the entry points into your patch code) public to other code fragments. Other functions, such as `Change_Depth()`, whose names you deleted from the .exp file, are now private to the code fragment.

Why did you build a PowerPC shared library, rather than a code resource? While the CodeWarrior IDE lets you build a native resource, doing so requires that you kludge a dummy `main()` function. (Code resources must have one entry point. The linker will demand one.) It's much easier to build the shared library, minus the default entry points that you removed via the PPC

PEF Panel. Unfortunately, there's a problem going the shared library route: You need to move the PowerPC code out of the file's data fork and into the resource fork, so it looks like a resource. Mathemæsthetic Inc.'s Resorcerer is a resource editor similar to ResEdit that lets you cut and paste between file forks. However, with CodeWarrior in hand, you can manage this chore ourselves. Start a new project, called "Klepto.μ", and type:

```
#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Memory.h>
#include <ToolUtils.h>
#include <StandardFile.h>
#include <Errors.h>
#include <Resources.h>

/* Various constants */
#define NIL             0L
#define FALSE           false
#define TRUE            true
#define DEFAULT_VOL     0
#define ONE_FILE_TYPE   1
#define POWER_PC_FRAG   'PPC '     /* Resource type */
#define FRAG_ID         300        /* Resource ID */

void Move_Fork(short input);
void main(void);

void Move_Fork(short input)
{
OSErr       fInputErr;
long        codeFragSize;
Handle      fragBuff;

   fInputErr = GetEOF(input, &codeFragSize); /* Get file length */
/* Enough memory? */
   if ((fragBuff = NewHandle(codeFragSize)) != NIL)

      {                       /* Read in fragment */
      if (!(fInputErr = FSRead(input, &codeFragSize, *fragBuff)))
         {                    /* Treat as a resource */
```

```
            AddResource(fragBuff, POWER_PC_FRAG, FRAG_ID, NIL);
            if (!ResError())   /* No trouble? */
               { /* Write frag to resource fork */
               WriteResource(fragBuff);
               if (ResError())
                  SysBeep(30);
               } /* end if !ResError */
            } /* !fInputErr */
         } /* end if != NIL */
      ReleaseResource(fragBuff);     /* Free the memory */
} /* end Move_Fork() */
```

The function `Move_Fork()` performs the operations necessary to copy the PowerPC code from the file's data fork to the resource fork of a new file. The file will already be opened by routines in `main()`. Let's see how this is done. First, you use the `GetEOF()` routine to obtain the size of the file's data fork. This size value is passed to `NewHandle()` to create a memory buffer large enough to hold the shared library. The File Manager routine `FSRead()` reads the code fragment into this buffer. With the PowerPC code in memory, you next call `AddResource()`. This routine creates a resource entry for this buffer within a file's resource fork. (Remember, this file must have been opened by a previous Resource Manager call.) You use `WriteResource()` to write the PowerPC code into the file's resource fork. Finally, you call `ReleaseResource()` to discard the memory used by `fragBuff`, because this buffer is now considered a resource by the Mac OS. To change the resource's type and ID number, you can edit the definitions for `POWER_PC_FRAG`, and `FRAG_ID`.

Now let's add `main()` where you open and close the files:

```
void main(void)
{
unsigned char  file name[21] = {"\pFlipDepth. .PPC.rsrc"};
 /* Output file's creator and type */
OSType             fileCreator = {'RSED'};
OSType             fileType = {'rsrc'};
OSErr              fileError;
short              inFileRefNum, outFileRefNum;
StandardFileReply  inputReply, outputReply;
short              oldVol;
/* File type for shared library */
SFTypeList         shlbType = {'shlb'};
CursHandle         theCursor;     /* Current pointer icon */
```

```
/* Lunge after all the memory you can get */
   MaxApplZone();

/* Make sure you have got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(NIL);

/* Open the input file */
   StandardGetFile(NIL, ONE_FILE_TYPE, shlbType, &inputReply);
   if (inputReply.sfGood)
      {
      GetVol (NIL, &oldVol);    /* Save current volume */
      if ((fileError = FSpOpenDF (&inputReply.sfFile, fsCurPerm,
                                  &inFileRefNum)) != noErr)
         {
         SysBeep(30);
         return;
         } /* end if error */

/* Open the output file */
      StandardPutFile("\pSave code fragment in:", file name,
               &outputReply);
      if (outputReply.sfGood)
         {
         SetVol(NIL, outputReply.sfFile.vRefNum);
         fileError = FSpCreate(&outputReply.sfFile, fileCreator,
                     fileType, smSystemScript);
         switch(fileError)
            {
            case noErr:
            break;
```

```
            case dupFNErr:   /* File already exists */
        if ((fileError = FSpDelete(&outputReply.sfFile)) == noErr)
                    {
                    if ((fileError = FSpCreate(&outputReply.sfFile,
                                               fileCreator,
                                               fileType,
                                               smSystemScript)) !=
                                noErr)
                       {
                       SysBeep(30);
                       FSClose(inFileRefNum);
                       SetVol(NIL, oldVol);
                       return;
                       } /* end if != noErr */
                    } /* end == noErr */
                 else
                    {
                    SysBeep(30);
                    FSClose (inFileRefNum);
                    SetVol(NIL, oldVol);
                    return;
                    } /* end else */
              break;   /* end case dupFNErr */
              default:
                 SysBeep(30);
                 FSClose(inFileRefNum);
      /* Close the input file */
                 SetVol(NIL, oldVol);
      /* Restore original volume */
                 return;
              } /* end switch */

/* Open file's data fork. Do this only to get a file ref number */
           if (!(FSpOpenDF (&outputReply.sfFile, fsCurPerm,
                    &outFileRefNum)))
              {
/* MUST create resource map in resource fork or no resource
  writing occurs */
              FSpCreateResFile (&outputReply.sfFile, fileCreator,
                               fileType, smSystemScript);
              if (!ResError())
                 {  /* Open resource fork */
                 FSpOpenResFile (&outputReply.sfFile, fsCurPerm);
```

```
            if (!ResError())
               {  /* Change cursor */
               theCursor = GetCursor(watchCursor);
               SetCursor(&**theCursor);
               Move_Fork(inFileRefNum);
               FSClose(outFileRefNum);
               SetCursor(&qd.arrow);
    /* Restore cursor */
               } /* end if !ResError */
            } /* end if !ResError */
         FlushVol (NIL, outputReply.sfFile.vRefNum);
         } /* end if !FSpOpenDF */
      } /* end if outputReply.sfGood */
    FSClose (inFileRefNum);
    SetVol(NIL, oldVol);
    /* Restore current volume */
  } /* end if inputReply.sfGood */

} /* end main() */
```

You'll notice there's no event loop in this program. That's OK, because the Standard File Dialog boxes have enough built-in smarts to manage most of the events required to make a file selection, and `Move_Fork()` manages all of the file I/O. That's all you need for this quick and dirty little program. Once you get past the initialization code, you can see most of the code came from the `Ask_File()` function in SonOMunger. I did remove all the error reporting calls, replacing them with `SysBeep(30)`. This simplifies the making of the application while still providing some measure of safety. This is *not* the sort of thing you would do for a program for public distribution, but for in-house work it's quite adequate.

The C code for picking and opening the input file remains the same as SonOMunger's, except that `StandardGetFile()` filters out all files but types of 'shlb'. The code for opening the output file is the same, up to a point. You first open the output file using `FSpOpenDF()`, only so that you can get a file reference number in order to close the file when you are done. Next, you call `FSpCreateResFile()` to create a resource reference map in the file's resource fork. If you fail to perform this step, no resource writing can be done to the file. The final step before calling `Move_Fork()` is to open the resource fork using `FSpOpenResFile()`. Note that the resource file routines report errors back through the `ResError()` function. These routines also don't use a file reference number. That's because once a link is established between the file and the

Resource Manager, it persists through all subsequent uses of the resource routines until the file is closed.

Because this code steals PowerPC code from a file's data fork, I named this source file "Klepto.c." Add it to the Klepto. project. Set the project's preferences as follows: in the project panel (either PPC or 68K) set the project type to an application (use the factory defaults other than to name the output file "Klepto."), compile, and make the application. That's right—you didn't build any resources with Rez in order to make ".Klepto." Because "Klepto" doesn't use any special resources, and the resources for Standard File Dialog boxes come from the System file, the program code runs as it is.

Double-click Klepto to launch it, and a Standard File Dialog box appears. The only file that should appear in the dialog is your shared library file, "FlipDepth.lib." Click on the Open button or press Return to choose the file. Immediately, a second Standard File Dialog box appears. The default output name for this dialog box is `FlipDepth. .PPC.rsrc`. Either click the Save button, or type in another file name and press Return. Klepto should quit shortly, leaving you a resource file with the given name. If you double-click this file, ResEdit launches, and you can examine the PPC resource to see the PowerPC code within it. When you make the 680x0 version of your Extension, you'll simply add this resource file to the project. CodeWarrior copies this file's resources—and thus the PowerPC code in it—to the resulting Extension file.

I'm sure you can build and link multisegmented code resources. However, I think you're going to have to explicitely add code to deal with this, per my description in the text. Because stand-alone resources execute in place without preparation by the Process Manager, you have to add the support code yourself.

While on the subject, it's now time to build the 680x0 project. Close the Klepto.µ project window, and create a project called FlipDepth.µ.68K, using the ~Min MacOS PPC C/C++.µ template. Add the file FlipDepth.c and remove the file "CPlusPlus.lib" from this project. Select Preferences from the Edit menu and choose the 68K Processor panel. Click the Code Model popup menu and select Small. In the 68K Linker panel, go to the Linker Info section and check the item Link Single Segment. Unless you're adding your own code that handles multiple code segments, the Small memory model and Link Single Segment settings are required. Optionally, in the Debugger Info section of this panel, you might want to uncheck the Generate A6 Stack

Frames item. The A6 stack frame code that's produced when this item is checked provides hooks for a debugger program. This isn't necessary for this example unless you want to experiment with a debugger on the Extension file. Next, pick the 68K Project panel. For the Project Type, click the popup menu and choose Code Resource. The panel's contents will change. In the Code Resource section, type **FlipDepth** for the output file's name, for the ResType item, type **INIT**, and for the ResID item type **128**. This sets up the resource's type and ID number. The last thing to do is set the output file's type and creator. Go to the Creator item and type **FLDP**, and for the Type item enter **INIT**. Finally, go to the Resource flags and click the popup menu. Check the System Heap and Locked items. These settings ensure that the Resource Manager loads the 68K code resource into the system heap, and locks it in place. Leave the Header Type item as Standard. Click the OK button to save the new settings.

Like SwitchBank, FlipDepth is going to have its own cool icon, which means you need some resources that describe the icon and other particulars. This information is stored as Rez source text in the file FlipDepth.r, which you should add to the project via the Add Files... selection on the Project menu. Double-click this file and examine some of the Rez code in the editor:

```
#include "SysTypes.r"
#include "Types.r"

/* Resource IDs for file refs & icons */
#define EXTENSION_FREF    128
#define BAD_LOAD_FREF    129
#define FLIPDEPTH_ICON    128
#define BAD_LOAD_ICON    129

resource 'BNDL' (128)
{
  'FLDP', 0,
  {
  'FREF', { 0, EXTENSION_FREF, 0, BAD_LOAD_FREF },
  'ICN#', { 0, FLIPDEPTH_ICON, 1, BAD_LOAD_ICON }
  }
};

resource 'FREF' (EXTENSION_FREF)
{
```

```
  'INIT',
  0,
  ""
};
resource 'FREF' (BAD_LOAD_FREF)
{
  'BADL',
  1,
  ""
};

/* Signature resource - all 'STR ' */
/* resources must be declared before this! */

type 'FLDP' as 'STR ';

resource 'FLDP' (0) {
   "FlipDepth 1.2"
};

data 'ICN#' (FLIPDEPTH_ICON) {
  $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
  $"6000 0006 4000 0002 8000 0001 83FF FFC1"
  $"83FF FFC1 8EAA AA81 8D55 5511 8EAA AA11"
  $"8D55 5411 8EAA A811 8D55 4011 8EAA 8011"
  $"8D55 0011 8EAA 0011 8D54 0011 8EA0 0011"
  $"8D40 0011 8E80 0011 8D00 0011 8000 0001"
  $"83FF FFC1 8000 0001 4E00 0002 6000 0006"
  $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
  $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
  $"7FFF FFFE 7FFF FFFE FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF 7FFF FFFE 7FFF FFFE"
  $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
};

data 'ICN#' (BAD_LOAD_ICON) {
  $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
  $"6800 0026 5C00 0072 8E00 00E1 87FF FFC1"
  $"83FF FFC1 8FEA AF91 8DF5 5F11 8EFA BE11"
```

```
  $"8D7D 7C11 8EBE F811 8D5F E011 8EAF C011"
  $"8D57 C011 8EAF E011 8D5E 7011 8EBC 3811"
  $"8D78 1C11 8EF0 0E11 8DE0 0711 81C0 0381"
  $"83BF FDC1 8700 00E1 4E00 0072 6400 0026"
  $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
  $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
};
```

Important

This listing doesn't include all of the data for the colored icons. For the complete listing, check Appendix C or the file on the CD. Like the SwitchBank Rez file, this is where the bundle resource (BNDL) and file reference resource (FREF) are defined. The ID numbers 128 and 129 tie things together. Resource ID 128 manages everything related to the file icon. The file icon also serves double-duty as the initialization success icon that `ShowInitIcon()` displays. ID 129 handles the failure icon only.

You need to pick up your PowerPC code resource in order to build the fat trap. This is just a matter of choosing Add Files... from the Project menu and selecting the file "FlipDepth.µ.PPC.rsrc." This file name should appear in the project window.

Now make the project. "FlipDepth.c" and "FlipDepth.r" will compile, and a "FlipDepth" file will appear, sporting your custom monitor icon. If the linker reports problems, double-check the 68K Linker panel and 68K Project panel settings. If the file sports a generic puzzle-piece Extension icon, make sure, in the 68K Project panel, that you've entered 'FLDP' as the file creator. Drag "FlipDepth" to the System Folder, and the Finder should request to place the file in the Extensions folder. Make sure that the Mac is set in a display mode other than its shallowest pixel depth, and reboot. When the Desktop appears, try typing Command-Shift-T. The screen should toggle to the shallowest pixel mode and then back when you hit the key sequence again.

Building a Fat Resource

You may have noticed something (or actually the lack thereof) when your Mac booted with the FlipDepth Extension. Namely, what happened to its success icon? The FlipDepth icon didn't appear during the boot process because you haven't added the code resource that contains the `ShowInitIcon()` function. This slipup illustrates the value of defensive coding. By performing a safety check for the presence of this particular resource, you avoided a crash, yet ensured that your patch code installed. Not all situations can be handled this gracefully, of course, but anything you can do to shield the user from a disastrous crash will be much appreciated.

With that lesson over, let's proceed to add `ShowInitIcon()` to the FlipDepth Extension. You'll be building another code resource that implements the function as both 680x0 and PowerPC code. Such a code resource is termed, not surprisingly, a *fat resource.* Before I can describe a fat resource, however, you have to back up a moment to explain the organization of stand-alone code resources.

A 680x0 stand-alone code resource typically starts with a sequence of bytes (called a *header*) that points to the start of executable code inside the resource. Put another way, the header contains a function pointer to the resource's `main()` function. This way when a program such as FlipDepth loads and calls a stand-alone resource (or an application such as Adobe Photoshop accesses a plug-in tool), the thread of execution gets routed into that resource's functions.

Important

Other types of stand-alone code resources, such as WDEF, CDEF, and DRVR, have a special header that encodes extra information. In DRVR, for example, this information describes whether the driver needs periodic service calls from the OS, and whether it responds to certain I/O requests. After this comes a set of function pointers that aim to the various I/O functions that the driver implements. The Mac OS understands the predefined format for a DRVR resource, and knows where in the header to pick up these function pointers. This isn't the case for the custom resources you write, so the resource header must start with a function pointer.

A fat resource has a predefined resource type of 'fdes' and both 680x0 and PowerPC code embedded inside it. It requires a special header that points both to the start of the resource's 680x0 code and to the start of its PowerPC code. That is, the header must have function pointers to both the 680x0 version of `main()` and to the PowerPC version of `main()` in the resource. If you're a real stickler for accuracy, the function pointers don't have to point to a `main()`. They can point to whatever function you like. However, because the linkers of most development tools expect you to tell them where executable code starts through use of the symbol `main`, you're better off writing code this way. The CodeWarrior PowerPC tools let you specify another function name for the start of executable code, but not the 680x0 tools. So, for consistency's sake and to avoid cluttering the code with a lot of conditional compilation statements, you'll stick with `main`.

So what does a fat resource header look like, exactly? It simply consists of a routine descriptor, followed by two routine records, and then two chunks of executable code. One routine record describes and points to the 680x0 code, and the other routine record describes and points to the PowerPC code. From your discussion of routine descriptors in Chapter 4, you thereby conclude that a fat resource starts with the A-trap word `_goMixedModeMagic` (hexadecimal 0xAAFE). When executed, the `_goMixedModeMagic` trap invokes the Mixed Mode Manager. The Mixed Mode Manager determines what ISA is currently active, and then uses the routine descriptor to locate the start of the appropriate code section inside the resource. Execution proceeds from that function.

This elegant mechanism works fine for Power Macs, but what if a 'fdes' resource is called on a 680x0-based Mac? Usually, the processor trips over an unimplemented instruction exception and crashes, because the `_goMixedModeMagic` trap word is undefined on these systems. I say usually, because if a 680x0-based Mac is running OpenDoc, the resource executes as described. That's because OpenDoc uses a 680x0 implementation of the Code Fragment Manager, which relies on the Mixed Mode Manager. Because OpenDoc isn't currently part of the system software, there's no guarantee that it, and the 680x0 Code Fragment Manager, will be present. In recognition that programmers want their code resources to work on the largest set of Macs, Apple devised a *safe fat resource* of type 'sdes.' A safe fat resource's header starts with some 680x0 glue code. The first time a program loads an 'sdes' into memory, it locks it, and then calls it. This glue code checks the

operating system version number and determines whether the Mixed Mode Manager is present. If it isn't, and the resource is executing on a 680x0 Mac, the glue code patches a function pointer into the resource header. Otherwise, it writes a routine descriptor into the header. Finally, the thread of execution jumps back into the modified header. While this design incurs some overhead on the initial function call, subsequent calls to the sdes resource execute rapidly, because now the resource has a standard header (either function pointer or routine descriptor) tacked onto it.

Background Info

You use the Rez tool to build a fat resource. A set of templates, located in the header file "MixedMode.r," directs Rez to combine 680x0 and PowerPC code blocks into a single resource, along with the appropriate header data. If you want to examine the structure of the 'fdes' and 'sdes' headers in excruciating detail, use the CodeWarrior editor to examine the code in the Rez header file "MixedMode.r." This file is located on the path CodeWarrior:MacOS Support:RIncludes. It's worth examining this code to learn how fat resources are put together. Also, you'll be using these templates to build the `ShowInitIcon()` resource.

ShowInitIcon Code

Now that you have learned what a fat resource is, it's time to examine the code that produces one. The `ShowInitIcon()` function is responsible for displaying an icon onscreen during the boot process. Recall that FlipDepth's `main()` calls this function to display one icon if the patch code installs successfully (the success icon), and another icon that indicates the installation, due to an error, aborted (the failure icon). `ShowInitIcon()` has a long history, starting as an idea by Steve Capps (one of the original authors of the Finder). It was initially written in assembly language by Paul Mercer, Darin Adler, and Paul Snively. `ShowInitIcon()` was converted to the C language by Eric Shapiro. It was later rewritten by Peter N. Lewis, Jim Walker, and François Pottier. Their version is presented here. Go to the ShowInitIcon folder inside the FlipDepth folder and open the file "ShowInitIcon.h." It consists of the following declarations:

```
#ifndef __ShowInitIcon__
#define __ShowInitIcon__

#include <Types.h>

// Usage: pass the ID of your icon family (ICN#/icl4/icl8) to have
//    it drawn in the right spot.
//    If 'advance' is true, the next INIT icon will be drawn to
//    the right of your icon. If it is false, the next INIT icon
//    will overwrite yours. You can use it to create animation
//    effects by calling ShowInitIcon several times with 'advance' //    set to
false.

#ifdef __cplusplus
extern "C" {
#endif

pascal void ShowInitIcon (short iconFamilyID, Boolean advance);

#ifdef __cplusplus
}
#endif

#endif /* __ShowInitIcon__ */
```

This header file contains the function prototype for `ShowInitIcon()`, and it sets up some compiler environment variables. The environment variables are useful if you're linking this code directly into your Extension. Otherwise, you can delete the ShowInitIcon.h file and drop the function prototype declaration seen here and the `#include` statement for the Types.h file into ShowInitIcon.c. This simplifies tracking these declarations, but keep in mind that ShowInitIcon.h works fine even if you're producing a code resource with ShowInitIcon.c. While on the subject, let's examine that file.

```
// ShowInitIcon - version 1.0.1, May 30th, 1995
// This code is intended to let INIT writers easily display an icon at // startup
time.

// This version features:
// - Short and readable code.
// - Correctly wraps around when more than one row of icons has
//      been displayed.
```

```
// - works with System 6
// - Built with Universal Headers & CodeWarrior. Should work with
//   other headers/compilers.

#define USESROUTINEDESCRIPTORS GENERATINGCFM

#include <Memory.h>
#include <Resources.h>
#include <Icons.h>
#include <OSUtils.h>
#include <LowMem.h>
#include "ShowInitIcon.h"

#define SYSTEM_7    0x0700

// You should set SystemSixOrLater in your headers to avoid
// including glue for SysEnvirons.

//    ------------------------------------------------------------
// Set this flag to 1 if you want to compile this file into a
//     stand-alone resource (see note below).
// Set it to 0 if you want to include this source file into your
//     INIT project.

#define STAND_ALONE_RESOURCE  1

#if STAND_ALONE_RESOURCE

#define ShowInitIcon main    // For the linker, which expects
                    // a "main" symbol

enum
{
  kShowInitIconInfo = kPascalStackBased
    ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
    ¦ STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(Boolean)))
};

ProcInfoType __procinfo  = kShowInitIconInfo;  // Resource argument
                             // info, is 0x180

#endif
```

Here's your familiar set of header files, followed by the definition (`STAND_ALONE_RESOURCE`) that determines whether the generated code behaves as a function to be linked into your code, or as a stand-alone resource. In the instance of stand-alone code, the conditional `if` sets up several items. First, the symbol `ShowInitIcon` is defined as `main` to assist the linker locate the start of executable code in the resource. Next, the size and type of `ShowInitIcon()`'s arguments are defined (via `kShowInitIconInfo` and `ProcInfoType`) in order to build a routine descriptor.

```
// ------------------------------------------------------------
// The ShowINIT mechanism works by having each INIT read/write
// data from these globals. The MPW C compiler doesn't accept
//  variables declared at an absolute address, so I use these
//  macros instead. Only one macro is defined per variable;
//  there is no need to define a
// Set and a Get accessor like in <LowMem.h>.

#define  LMVCoord    (*(short *) (LMGetCurApName() + 32 - 6))
//#define  LMVCoord    (* (short*) 0x92A)
#define  LMVCheckSum    (* (short *) (LMGetCurApName() + 32 - 8))
//#define  LMVCheckSum    (* (short*) 0x928)
#define  LMHCoord    (* (short *) (LMGetCurApName() + 32 - 4))
//#define  LMHCoord    (* (short*) 0x92C)
#define  LMHCheckSum    (* (short *) (LMGetCurApName() + 32 - 2))
//#define  LMHCheckSum    (* (short*) 0x92E)

// ------------------------------------------------------------
// Prototypes for the subroutines. The main routine comes first
// (in ShowInitIcon.h); this is necessary to make THINK C's
// "Custom Header" option work.

static unsigned short CheckSum (unsigned short x);
static void ComputeIconRect (Rect* iconRect, Rect* screenBounds);
static void AdvanceIconPosition (Rect* iconRect);
static void DrawBWIcon (short iconID, Rect *iconRect);
```

`ShowInitIcon()` reads screen position information from a low memory global, `CurApName`. This is so that each Extension knows where to plot its icon without overwriting an existing one. The Mac OS uses `CurApName` to hold the name of the foreground application and is a 32-byte Pascal string. Because this global

is used occasionally during the boot process, you only tinker with the last four bytes in the string. These macros assign a descriptive name to each of the bytes and their address. Two bytes (`LMVCoord` and `LMHCoord`) hold the vertical and horizontal screen coordinates, while the other two bytes, `LMVCheckSum` and `LMHCheckSum`, hold running checksums to ensure that the coordinate data hasn't been corrupted.

On the off-chance that a later version of System 7.5.x might meddle with the location of some low memory globals, I rewrote these macros to use the accessor function `LMGetCurApName()`. The offsets help show where in the string the data is stored. These offsets are reduced to constants by the optimizer in the Metrowerks compiler, so that there's no performance penalty for making the code readable.

The various function prototype come next. Now, at last, comes the `main()` function.

```
// ------------------------------------------------------------
// Main routine.

typedef struct {
  QDGlobals  qd;        // Storage for the QuickDraw globals
  long       qdGlobalsPtr;  // A5 points to this place;
                    // it will contain a pointer to qd
} QDStorage;

pascal void ShowInitIcon (short iconFamilyID, Boolean advance)
{
  long       oldA5;    // Original value of register A5
  QDStorage  qds;      // Fake QD globals
  CGrafPort  colorPort;
  GrafPort    bwPort;
  Rect       destRect;
  SysEnvRec  environment;  // Machine configuration.

  oldA5 = SetA5((long) &qds.qdGlobalsPtr)
        // Tell A5 to point to the end of the fake
                              // QD Globals
  InitGraf(&qds.qd.thePort);
       // Initialize the fake QD Globals

  SysEnvirons(curSysEnvVers, &environment);
      // Find out what kind of machine this is
```

```
  ComputeIconRect(&destRect, &qds.qd.screenBits.bounds);
          // Compute where the icon should be drawn

  if (environment.systemVersion >= SYSTEM_7 &&
    environment.hasColorQD) {
    OpenCPort(&colorPort);
    PlotIconID(&destRect, atNone, ttNone, iconFamilyID);
    CloseCPort(&colorPort);
  }
  else {
    OpenPort(&bwPort);
    DrawBWIcon(iconFamilyID, &destRect);
    ClosePort(&bwPort);
  }

  if (advance)
    AdvanceIconPosition (&destRect);

  SetA5(oldA5);       // Restore A5 to its previous value
}
```

Here's where all the magic happens. First, define a storage area, `qd`, for your QuickDraw globals. Unlike building an application, where the development software automatically sets up the QuickDraw global storage (as described in Chapter 3), you must do this yourselves. Now comes the executable code. The first thing `ShowInitIcon()` does is save the current value in register A5 (680x0 processor or 68LC040 emulator) using the macro `SetA5()`. This macro also sets A5 to point to your `qd` globals. You are creating a temporary *stand-alone A5 world* for QuickDraw to use when it draws your icon.

You might be cringing at this point because you are messing with register A5. Earlier in this chapter, didn't I mention that that tampering with A5 was a bad idea? It is, but because you need QuickDraw to display your icon and QuickDraw is so intimately tied to the use of A5, you must take this step. The purpose here is to preserve the QuickDraw globals that the boot process uses, while still drawing your icon. Once `ShowInitIcon()` quits, your `qd` globals get scrapped, along with the `ShowInitIcon()` code. Recall also that the CodeWarrior IDE's 680x0 compiler generates stand-alone code such that any global variable access relies on register A4. This eliminates any conflicts

between your Extension and the boot process as far as register usage goes. This code shows you how to safely make use of QuickDraw in your Extension, if it is necessary. You can even go as far as using windows and dialog boxes. At boot time, for example, the AppleShare Extension uses a dialog box that prompts you for a password to a mount a server volume.

Hazard

For human interface reasons, it's not a good idea to interrupt the boot process to ask for information. This stalls the system startup until the user enters data, which might be a long wait if she has wandered off for a cup of coffee. No matter how good it is, this sort of behavior is guaranteed to get your Extension yanked if it hangs the unattended restart of a Mac after a power failure. This is especially true if that Mac functions as a file server or as a remote access node.

Now, initialize QuickDraw (`InitGraf()`) and call `SysEnvirons()` to check out the system. Next, call `ComputeIconRect()`, passing it the global `qd.screenBits.bounds`, which contains the dimensions (in pixels) of the main screen. `ComputeIconRect()` uses this information, and the data stored in `CurApName`, to position the 32- by 32-pixel rectangle that frames your icon. The code then checks to see whether the Mac is running System 7 and has a color display. If so, open a color grafport and use the Toolbox call `PlotIconID()` to draw FlipDepth's color icon. If you are running on a Mac without a color display, open a black-and-white grafport and call the custom function `DrawBWIcon()` to draw the icon. With that done, the function `AdvanceIconPosition()` is called. It updates the coordinate data in `CurApName`, so that `LMVCoord` and `LMHCoord` point to a new screen position for the next icon. If you're presenting a series of icons for an animation, call `ShowInitIcon()` repeatedly with the `advance` argument set to FALSE. In this case, `AdvanceIconPosition()` isn't executed, and each icon appears one on top of the other at the same screen location. When you complete the animiation sequence, the last call to `ShowInitIcon()` should be made with `advance` set to TRUE to bump the screen position for the next Extension.

```
// ----------------------------------------------------------
// A checksum is used to make sure that the data in
// there was left by another ShowINIT-aware INIT.
```

```
static unsigned short CheckSum (unsigned short x)
{
  return ((x << 1) ¦ (x >> 15)) ^ 0x1021;
}

// ------------------------------------------------------------
// ComputeIconRect computes where the icon
// should be displayed.

static void ComputeIconRect (Rect* iconRect, Rect* screenBounds)
{
// If you are first, you need to initialize the shared data.
  if (CheckSum(LMHCoord) != LMHCheckSum)
    LMHCoord = 8;
  if (CheckSum(LMVCoord) != LMVCheckSum)
    LMVCoord = screenBounds->bottom - 40;

// Check whether you must wrap
  if (LMHCoord + 34 > screenBounds->right) {
    iconRect->left = 8;
    iconRect->top = LMVCoord - 40;
  }
  else {
    iconRect->left = LMHCoord;
    iconRect->top = LMVCoord;
  }
  iconRect->right = iconRect->left + 32;
  iconRect->bottom = iconRect->top + 32;
}

// AdvanceIconPosition updates the shared global variables
// so that the next extension will draw its icon beside ours.

static void AdvanceIconPosition (Rect* iconRect)
{
  LMHCoord = iconRect->left + 40;  // Update the shared data
  LMVCoord = iconRect->top;
  LMHCheckSum = CheckSum(LMHCoord);
  LMVCheckSum = CheckSum(LMVCoord);
}
```

Here are the functions that manage the running checksum (`CheckSum()`) and the icon screen position (`ComputeIconRect()` and `AdvancePosition()`). `ComputeIconRect()` first verifies the stored screen coordinates by passing `LMVCoord` and `LMHCoord` to `CheckSum()` and then compares these values to those held in `LMVCheckSum` and `LMHCheckSum`. Notice the watchdog code that handles the default screen coordinate setup if the checksums fail to pass muster. This situation should only occur if, by some miracle, your Extension happens to execute first. (Extension files are loaded and executed by the Mac OS in alphabetical order.) In this case, `ShowInitIcon()` sets up the initial coordinates of the icon's rectangle 8 pixels, horizontally and vertically, from the bottom left corner of the screen.

If you're puzzling over the value of -40 for the vertical displacement, here's the score. Negative displacements move the current plotting position upwards or to the left in QuickDraw's coordinate system. Because QuickDraw plots rectangles starting from their upper left corner, and icons are 32 pixels wide, then the bottom of the icon appears 8 pixels above screen's bottom.

If an icon's rectangle comes within 34 pixels of the right of the main screen, more watchdog code wraps the rectangle's location by moving it back to 8 pixels from screen's left side, and up another 40 pixels. `AdvanceIconPosition()` simply bumps the icon rectangle 40 pixels to the right, computes new checksums based on these coordinates, and updates the locations in `CurApName`.

```
// DrawBWIcon draws the 'ICN#' member of the icon family.
// It works under System 6.

static void DrawBWIcon (short iconID, Rect *iconRect)
{
  Handle    icon;
  BitMap    source, destination;
  GrafPtr    port;

  icon = Get1Resource('ICN#', iconID);
  if (icon != NULL) {
    HLock(icon);
// Prepare the source and destination bitmaps.
    source.baseAddr = *icon + 128;        // Mask address.
    source.rowBytes = 4;
```

```
    SetRect(&source.bounds, 0, 0, 32, 32);
    GetPort(&port);
    destination = port->portBits;
                        // Transfer the mask.
    CopyBits(&source, &destination, &source.bounds, iconRect,
          srcBic, nil);
                        // Then the icon.
    source.baseAddr = *icon;
    CopyBits(&source, &destination, &source.bounds, iconRect,
          srcOr, nil);
  }
}
```

The function `DrawBWIcon()` supports Macs running System 6. Because Power Macs require System 7.1.2 or later (7.5.2 for second-generation PCI-bus Power Macs), this code probably never runs. It serves, however, as a terrific example of how to use the Toolbox routine `CopyBits()`, which is often used to draw custom images or dialog controls onscreen. These same principles can be applied to color icons and images, because `CopyBits()` is a color-capable routine.

As usual, `DrawBWIcon()` does a safety check for the presence of the black-and-white icon's ICN# resource. If it's present, you set up a handle (`icon`) that points to the icon's mask. An ICN# resource typically starts with an icon's image data and is followed by its mask data. Because a 32- by 32-bit icon consists of 128 bytes, the starting (or base) address of the mask's data is 128 bytes past the icon data (`*icon + 128`). Next, you indicate that there are 4 bytes of data per row (`rowBytes = 4`) in the icon's image. Last, set up the drawing rectangle's size for the source image, and arrange the current grafport as the destination for the drawing. The first call to `CopyBits()` uses the icon's mask data and QuickDraw's *bit clear* drawing mode (`srcBic`) to punch an appropriately shaped hole in the Desktop's background pattern. Then, adjust `icon` to point toward the actual icon data. A second call to `CopyBits()` uses the *or* drawing mode (`srcOr`) to drop the icon's image into this hole without disturbing the background pattern.

Compiling the Fat Resource

Now that you have toured the code, it's time to generate the resource. Open the project file "ShowInitIcon.µ.PPC." You'll see a project window, as shown in Figure 5.14.

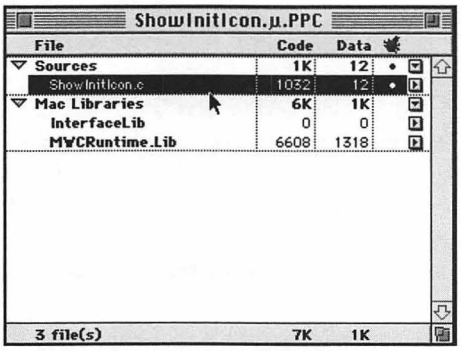


Figure 5.14 The PowerPC version of the ShowInitIcon project.

Check the preference settings for the project. The C/C++ Language and C/C++ Warnings should be the same as those in previous projects. Under the PPC Processor panel, ensure that the data structure alignment popup menu is set to 68K. For the PPC Linker panel, make sure that the Entry Point's Initialization and Termination items are empty. For the Main item, type in `main` (no leading underscores). If you weren't making a fat code resource, you could do away with the `#define` statement in ShowInitIcon.c, and type `ShowInitIcon` instead for the Main entry point. For the PPC PEF panel, make sure that the item Expand Uninitialized Data is checked.

Hazard

Unless you want to spend a lot of time debugging crashes, you must check this setting in the PPC PEF panel! Here's why. The PowerPC linker normally compacts any initialized data structures that the code fragment's data area contains. Nor does it allocate space for the uninitialized data variables in this area. These actions can conserve disk space for the code fragment file, particularly when the fragment deals with large data structures.

Recall in Chapter 4 that the Code Fragment Manager prepares a fragment for execution. Glance back at the installation code for FlipDepth, and you'll see that you used `GetMemFragment()` to load your code fragment into memory and prepare it. One of the Code Fragment Manager's jobs is to create the run-time links for the fragment's globals and functions; another

is to build the fragment's memory-resident data area. It does this by decompressing any initialized data structures and stashing these objects in the data area. It also allocates storage for the data area's uninitialized variables and zeroes this section of memory.

Stand-alone code resources, even when they're PowerPC code fragments, are loaded into memory and executed in place. Because the fragment isn't prepared for execution, any initialized data structures it has can't be compacted. Also, there must be file space allocated for all of the variables, even the unused ones. Now when the file image gets loaded into RAM, this arrangement creates the appropriately sized area in memory for these variables.

Important

The native patch code in FlipDepth's 'PPC ' resource is a *private* resource. The native code resource for `ShowInitIcon()` is an *accelerated* resource. The difference between the two is that a private resource acts more as a shared library, while an accelerated resource is just a code resource composed of PowerPC code.

The distinction is important. A private resource can have its own interface and multiple entry points. Witness FlipDepth's own patch code. It has two entry points, one for each of the patched Toolbox routines. An accelerated resource, on the other hand, must operate within the bounds of a code resource. As you've just seen in the hazard box, it can't rely upon services that the Code Fragment Manager provides. Furthermore, an accelerated code resource has only one entry point, although you can write the code such that a selector function calls different functions inside the resource.

Why would you write an accelerated resource? One reason is that it enables you to rapidly port existing 680x0 code resources to PowerPC code. The same expertise used to write code resources for 680x0 Macs can be applied to write native code resources. Another reason to write programs this way is that it enables you to use one code base that works for either processor.

The final stop for your PowerPC project's preferences is the PPC Project panel, as shown in Figure 5.15.

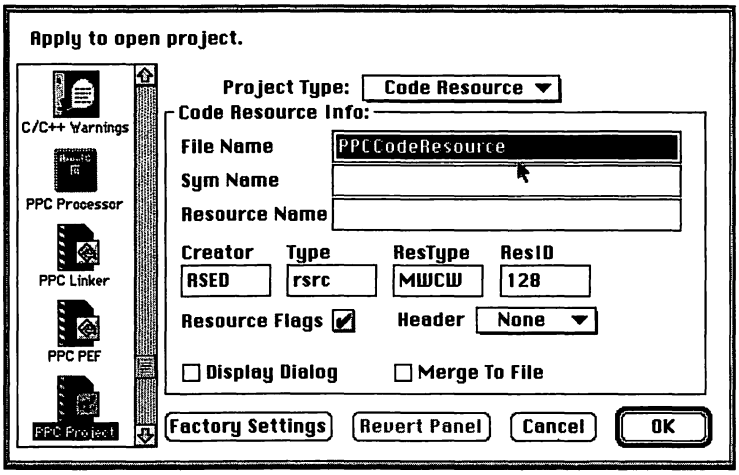


Figure 5.15 The PPC Project settings to make a native code resource.

There are several things to note here. The Project Type is set to Code Resource in the popup menu, the output file name is "PPCCodeResource," the resource's Type is 'MWCW', and its ID number is 128. It's important that you use these values, because the Rez file that builds the fat resource relies on this information. For this same reason, the Header popup menu item is set to None, because Rez builds the resource header later.

If everything is in order, build the project, and the output file "PPCCodeResource" should appear.

Now, it's time to make the 680x0 version of the resource. Open the project file "ShowInitIcon.μ.68K." The project window is shown in Figure 5.16.

As with the PowerPC version of the project, the settings are crucial. They're going to resemble the settings you used to produce the 680x0 version of FlipDepth, but they're worth reviewing again. In the 68K Processor panel, set the Code Model item to Small in the popup menu and the Struct Alignment popup menu to 68K. The 68K Linker panel, check the Link Single Segment and Merge Compiler Glue into Segment 1. The 68K Project window should appear as shown in Figure 5.17.

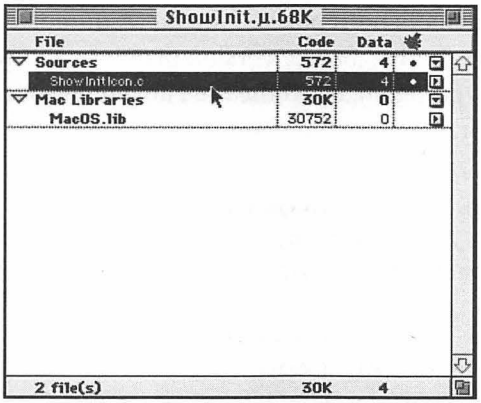


Figure 5.16 The 680x0 project window for ShowInitIcon.

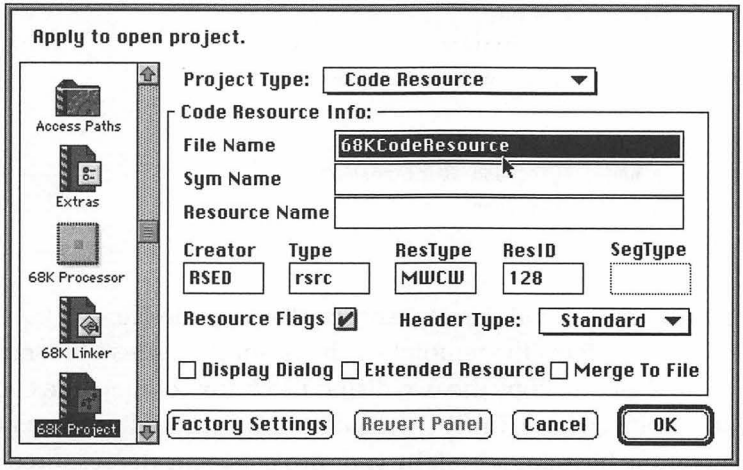


Figure 5.17 Creating the 680x0 code resource for ShowInitIcon.

Again, the Project Type is set to Code Resource. The name of the output file name and the resource type and ID should correspond to what's shown here. Leave the Header Type chosen as Standard, because either the 680x0 processor or the 68LC040 emulator expects a code resource to have a standard header. Leave the Resource Flag settings alone, because these are set when the fat resource is built. Make the project and the output file 68KCodeResource appears.

Now that you have made both parts of the fat resource, it's time to put them together. You'll use the ToolServer version of Rez to accomplish this. To understand what's going on, open the file FatCodeResource.r with the CodeWarrior editor.

```
#include "MixedMode.r"
// Enter the ProcInfo type as a hexadecimal value,
// as either $01, or 0x01. Use the CodeWarrior
// disassembler to determine this value

// Use resource type 'fdes' for a fat resource
// (PPC and 6K Macs running Mixed Mode)
// Use resource type 'sdes' for a safe fat resource
// (runs on all Macs)

resource 'sdes' (128, sysheap, locked) {
  $180,          // 68K ProcInfo
  $180,          // PowerPC ProcInfo
  // Specify file name, type, and ID of resource
  //   containing 68K code
  $$Resource("68KCodeResource", 'MWCW', 128),
  // Specify file name, type, and ID of resource
  //   containing a PEF container
  $$Resource("PPCCodeResource", 'MWCW', 128),
};
```

These Rez commands build the fat resource. The include file, "MixedMode.r," contains the templates (either 'fdes' or 'sdes') that create the fat resource header and copy the two disparate chunks of machine code into a single file. As you can see, the Rez commands in FatCodeResource.r are hard-wired for the file names built by your projects, plus the resource's type and ID.

The `ProcInfoType` value, which becomes part of the routine record that describes each code resource, is missing. Because `ShowInitIcon()` is a Pascal-based stack function with two arguments and doesn't return a result, its `ProcInfoType` is 0x180. If you're writing a different function, you can readily obtain its `ProcInfoType` value. After you've compiled the code, invoke the Disassemble command from the Project menu, and search the resulting dump window for the symbol `__procinfo`, as shown in Figure 5.18.

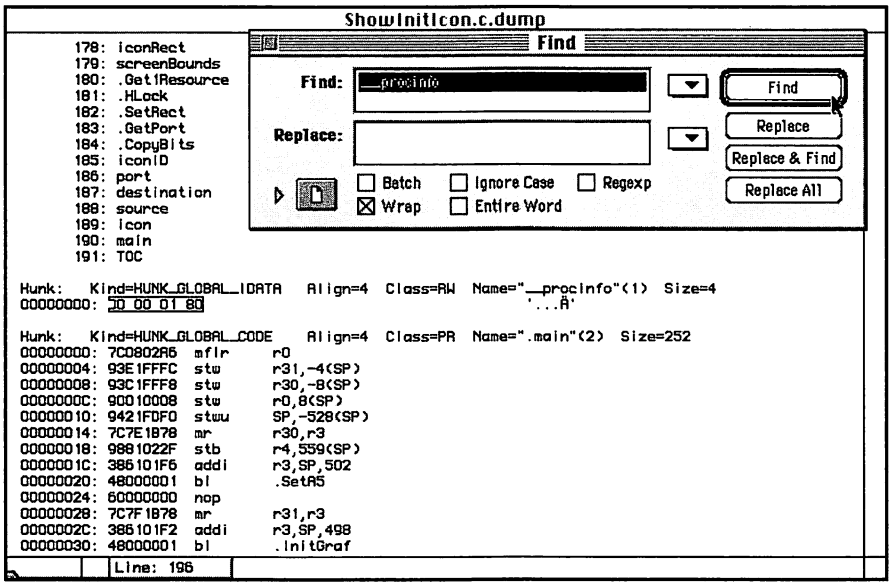


Figure 5.18 Getting the `ProcInfoType` for the `ShowInitIcon()` function. This value is the hexadecimal number associated with the symbol `__procinfo`.

The hexadecimal value linked to this name is the `ProcInfoType` value you type into the appropriate slot in FatCodeResource.r. Because this file already has the proper values for `ShowInitIcon()`, it's time to start ToolServer to build the code resource. Choose Start ToolServer from the CodeWarrior IDE's Tools menu. Select ToolServer Tools from the ToolServer menu, followed by Rez... from the hierarchial menu. When the Rez options window appears, go the the Rez Output File area and type **`rsrc`** in the Type item, press Tab to select the Creator item, and type **`RSED`**. Next, go to the popup menu with the default file name Rez.out and select Write Output to a New File from this menu. A Standard File dialog box appears. Check that you're in the FlipDepth:Projects folder. Type the name **`ShowInitIcon.rsrc`** and press Return. With the output file's name, type, and creator chosen, let's specify the input. Start by clicking the Files & Paths button. Rez places another window titled Files & Paths... on the screen. Now, point Rez to the location of your code resources and click the Include Paths... button (*not* the #Include Paths... button). A modified Standard File dialog box appears. Check that the current directory is the FlipDepth:ShowInitIcon folder. If so, click the Add Current Directory button, and this pathname is appended into a list at the bottom of the window. Click the Done button, and you're returned to the Files & Paths window.

Now, go to the Redirection item in this window and click the Input popup menu. Select Existing File from this menu, and a Standard File dialog box appears. Choose FatCodeResource.r, and press Return. Finally, click the Continue button to exit the Files & Paths window. Double-check all the settings, and then click the Rez button. You should hear some hard disk activity, and the status pane indicates that Rez is active. When the cursor changes from the rotating beach ball busy indicator back to an arrow, click the WorkSheet window. Switch to the Finder and check that the file ShowInitIcon.rsrc was created. If the file exists, go back to the CodeWarrior IDE and stop the ToolServer.

At long last, you are ready to add the `ShowInitIcon()` fat resource to the FlipDepth Extension file. There are two ways to do this. You can double-click the ShowInitIcon.rsrc file to launch RezEdit, because you set the file's creator to this utility. You can use ResEdit to cut and paste the 'sdes' resource from this file into "FlipDepth." Or, because the CodeWarrior IDE is still active, you can open the FlipDepth.µ.68K project file, select Add Files... from the Project menu. Navigate the Standard File dialog box into the FlipDepth:Projects folder, and add the file "ShowInitIcon.rsrc" to this project. Build the project again. The Toolbar should indicate that the file was copied to the output file. Drag the new FlipDepth Extension to the System Folder, reboot your Mac, and you should see FlipDepth's monitor icon appear when the Extension files load.

The nice thing about this design is that it's modular. If a new version of `ShowInitIcon()` should appear on the Internet, you can easily update FlipDepth's icon display without recompiling the Extension's source code.

You might be wondering if FlipDepth itself can be made into a fat resource. The answer is no, for a number of reasons. The major reason is the asymmetry between the 680x0 and the PowerPC versions of the patch code for the `PostEvent()` OS trap. Because it is a register-based routine, you had to resort to assembly language and a batch of conditional compilation statements to make both versions of the patch code. If you had used a different event-gathering routine, such as `SystemEvent()`, which is stack-based, you could have written all your your code C and gone the fat resource route. The other major problem is that the glue code in safe fat resource headers modifies the 680x0 registers A0, D0, and A1—the very same registers that an OS Toolbox routine uses to receive arguments and return results.

Summary

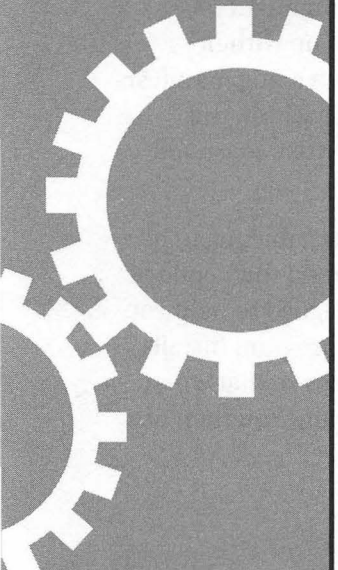
In this chapter, you have seen how to apply the knowledge you've gained about the PowerPC run-time architecture to solve specific programming problems, especially to guarantee an orderly switch from one instruction set to another when calling your custom function. As you have walked through the code of these two programs, you can see that this isn't difficult. Furthermore, it should be obvious that access to the global data of any program and OS Toolbox routines is far simpler on a PowerPC system than it is with a 680x0 system. This goes a long way toward helping developers write more Power Macintosh software. Finally, you have seen how to build fat resources, which are the fundamental building blocks for any cross-processor application.

Now that you have developed your programs, let's get to the other stuff—debugging.

The Art of Debugging

The material in this chapter will be of no interest to those programmers who write perfect programs, every time.

Seriously though, it is inevitable that program code has bugs. Programming is where you give the computer precise directions in what amounts to a second language for you. Despite C's elegant terseness of syntax (or because of it), there's the inevitable conversational misstep that causes the Mac to freeze up like a social misfit at a debutante's ball. In this chapter you'll look at the high-level debugging tools CodeWarrior provides in the form of MW Debug; Apple's low-level debugger, MacsBug; and The Debugger—a hybrid low- and high-level debugger. The chapter finishes with some common sense debugging techniques. Keep in mind that the PowerPC versions of these tools are changing rapidly, and some features and capabilities may differ from what you see here.



Important
The text of this book was written using the full version of Metrowerks CodeWarrior. You'll have to use slightly different steps when using CodeWarrior Lite from the CD. The commands New, New Projects, and Add File... are not available. Because of these limitations, it can only work with the sample files provided on the CD.

So, if you are following along using CodeWarrior Lite, when the text tells you to use the New, New Project..., or Add File... commands, you should instead open the related project file and keep it open throughout the exercise. All the associated files will already be in the project, so you won't need the Add File... command. Then, you can follow the same procedures as if you were using the full version of CodeWarrior.

About Debuggers

You've just completed writing that next killer application that users will flock to, with their wallets open. The code passes muster with the CodeWarrior IDE's C compiler, and after a few minor revisions the linker approves, too. But when you launch the application, either from within CodeWarrior or by double-clicking the resulting file, you get the infamous "bomb box," complete with a sizzling bomb icon. This dialog box is produced by the System Error Handler, which the operating system calls when it detects a fatal error or exception. This assumes that the cause of the error hasn't seriously trashed the operating system in the process. In that case, you might be spared the pyrotechnics, and the Mac instead simply seizes up with no warning at all. Despite this, consider yourself lucky that such a bug manifests itself so rapidly. It's those slowly ticking logic bombs lurking within the program code that go off minutes or hours later which can drive seasoned programmers to drink—and I don't mean Jolt cola, either.

No matter what type of program bug it is, or how long it takes the bug to bite, programmers rely on their wits, intuition, and debuggers to rid their code of these pests. A debugger is a highly specialized program designed to help you track down program bugs, hence the name. The debugger program installs its own exception handlers or uses advanced system routines so that when an exception occurs, it can seize control and halt the program. You then use

the debugger to investigate the exception's cause by examining the program's variables and data structures. If necessary, you can have the debugger take charge at designated points in the program, and single-step through the program's instructions, tracing the path of execution up to the crash. These debugger features enable you to reconstruct the crash scene. This usually gives you a good idea of where the bug is and how to fix it.

Background Info

It's nomenclature time again. Debuggers generally fall into two categories: hardware and software. A hardware debugger uses dedicated hardware to perform the debugging process, and a software debugger is a special computer program.

A typical hardware debugger is an In Circuit Emulator, or ICE for short. As its name implies, an ICE is a dedicated set of hardware that connects in-line with the test computer's processor, or replaces the processor entirely with custom circuitry. Special software lets you halt a program's execution based on hardware accesses, such as read/write operations to a memory location or an I/O port. Such fine control is possible because the ICE hardware eavesdrops on the bus signals and detects when a bus access touches the memory locations you request. An ICE is not usually necessary for development at the application level. It's used by the hardware and firmware designers as they build the prototype computer system and its ROM code. Because you are debugging programs here, not building a computer, this is the last mention of hardware debuggers.

Software debuggers are used to debug applications or software components such as plug-in modules or stand-alone code resources. These software debuggers may be further subdivided into two categories: low-level and high-level. A low-level debugger operates by using as few of the operating system resources as it can. Because of this, these debuggers are very robust. They continue to function even though a buggy application may have done heavy damage to the operating system. On the other hand, such debuggers typically have a minimalist interface and display. You can examine the program, but usually only as machine code instructions, and you need to know memory addresses of a program variable to view its contents. Apple's MacsBug is an example of a low-level debugger.

continues

continued

High-level debuggers rely heavily on the operating system to provide services such as windows and menus. In turn, they provide an easier to use interface and a sophisticated display. They can show a test program's code as either source or assembly language statements. Variables can be monitored simply by knowing the variable's name, not its memory location. Their values can be displayed in a variety of formats. On the other hand, because these capabilities depend on the operating system's health, substantial damage to it by a program error causes high-level debuggers to go down in flames along with the buggy program. Another limitation is that you can't debug certain types of code, such as MDEF (menu definition handlers), completion functions, or interrupt tasks. That's because some of these code types function on the fringes of the operating system (such as the interrupt task), and others pose reentrancy problems (you can't debug a new menu handler when the debugger itself uses menus). CodeWarrior's MW Debug is an example of a high-level debugger.

Despite these limitations, a high-level debugger is a good way to confirm that a program works as it should. Also, it's very good at quickly locating the vicinity of the problem code, which helps reduce the time it takes to zero in and fix the error. Also, a low-level debugger requires that you learn a lot of details about the processor, the operating system, and the compiler's output before you can make sense of what's going on. In short, a low-level debugger has a steep learning curve, whereas high-level debuggers only require that you know the programming language.

Both types of debuggers let you step through the statements one line at a time, or set control points called breakpoints. A breakpoint marks the instruction where the program's path of execution breaks (or halts) out of its normal course, and the debugger program takes control. Breakpoints thus allow you to run a program up to a suspect location. You can examine critical program variables and begin single-stepping from the breakpoint location to gather additional information.

Another valuable capability these debuggers offer is another control feature called a watchpoint. A watchpoint is similar to a breakpoint, but instead of halting the program flow on the execution of a particular instruction, the triggering event is a change in a data variable. For example, you might

monitor the value in a key global variable and halt the program if it changes. Or, you might keep watch on a memory block in your application heap that seems to get inexplicably hammered by your program.

So far, you have been reading about debuggers that run on the one target machine. There is another category of debugger here: a two-machine debugger. A two-machine debugger uses a small code "nub" (a control program) on the target machine, while the debugger itself runs on a different machine (called the host). The host machine communicates to the nub on the target via a wire, typically a serial cable. The big advantage to a two-machine debugger is that the host can support a high-level front end, while the low-level nub can usually survive the target machine's operating system being destroyed. A two-machine debugger can also provide source-level debugging for virtually any code in the target system. The big disadvantage is that this type of debugger requires two machines. Apple's initial PowerPC debugger, Macintosh Debugger for PowerPC, was a two-machine debugger. It has been modified recently to operate as a single machine debugger.

The concept of a debugging code nub also exists for single-machine debuggers. For example, CodeWarrior has a "MetroNub" file that establishes low-level communications and control between your test program and MW Debug.

As a high-level debugger, MW Debug can be used to single-step through the source code, and set breakpoints. It also displays the contents of variables, and lets you change their value. This way, as you step through the program, you observe what the code is actually doing and what values it's working with. By changing the values of function results, you can force the program through an error handler to check the application's robustness. MW Debug also allows you to examine a program as assembly language instructions.

Currently MW Debug can debug applications, shared libraries, and certain code resources. If, however, you're writing interrupt-level code, or code resources that can create reentrancy problems, you'll have to use a low-level debugger. Examples of code resources that have reentrancy issues are MDEF (menu definition resource), CDEF (control definition resource), and LDEF

(list definition resource). Another hard-to-debug code type is the Drag and Drop callback functions, because the Drag and Drop mechanism operates at interrupt time.

Using the CodeWarrior Debugger

In order for MW Debug to display variables and trace through the source code, it requires specific information about your program. You supply this vital data by preparing the program for debugging in the Metrowerks IDE. This preparation involves only a few changes to the project's preferences settings, and simply recompiling the program to make a new version. Along with the new executable application file, CodeWarrior also generates a symbolics, or symbols file. This symbols file contains the names of the variables and functions used in your program, plus their location in both the source code file and in the application file. MW Debug uses this symbol file information to manage the debugging session.

The symbols file CodeWarrior makes has the same name as the application's name, plus an extension of .SYM for the 680x0 code, and .xSYM for the PowerPC code. For example, let's assume you compile the source file in project "Klepto.μ" for debugging, and name the application Klepto. The CodeWarrior IDE generates a file named "Klepto.SYM" for the 680x0 version of the program, and "Klepto.xSYM" for the PowerPC version.

Let's take the "SwitchBank" program and run through parts of it with MW Debug. First, go to the folder labeled SwitchBank (debug). Now launch CodeWarrior by double-clicking on the "SwitchBank.μ.PPC" project file. Go to the Project window, and in "SwitchBank.c" file slot, click on the area beneath the bug icon. A small dot appears (see Figure 6.1). This dot is the Generate SYM Info marker. Now whenever the linker generates an output file, it creates the required symbolic debug information for the marked file. You can choose one or more files for debugging.

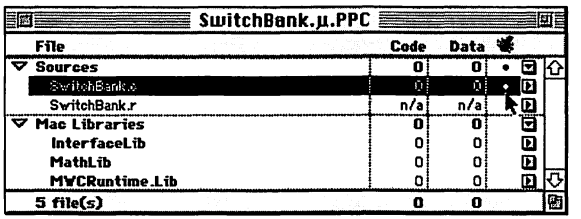


Figure 6.1 Marking a file for debug output.

Now that you've marked the source file, you need the CodeWarrior linker to actually generate the debug information. Select Preferences from the Edit menu, and select the PPC Linker Panel. Under the Link Options section, click on the Generate SYM File item to check it. Also check the Use Full Path Names item (see Figure 6.2). The Full Path Names has the linker generate a complete path specification for a file, such as Tachyon:CodeWarrior:Code Examples PPC:SwitchBank (debug):SwitchBank.c. While checking this item isn't necessary, it helps MW Debug locate the files it needs, especially if they're located somewhere other than the CodeWarrior folder or the project folder. If you're using the CodeWarrior IDE to produce 680x0 code, go to the 68K Linker Panel and check the Generate A6 Stack Frames item.

Remaking the application is the last step in the preparation sequence. Choose Make from the Project menu or type ⌘-M. CodeWarrior first recompiles the source, and then the linker produces the application file and the symbols file.

At this point, you have two options. If you're tight on memory, quit the CodeWarrior IDE and manually launch MW Debug, as described next. If you've got lots of spare memory, you can launch both MW Debug and your program from within the CodeWarrior IDE.

To do this, select the Enable Debugger item from the CodeWarrior IDE's Project menu. You'll notice that the Run item in this menu has changed to Debug. Now when you run the program (either by selecting Debug from this menu or clicking on the Run icon in the Toolbar), MW Debug automatically starts. Starting MW Debug this way allows you to do a quick program check-out, with the option to drop back into the CodeWarrior IDE if you spot a problem. Note that this feature is for use with applications only.

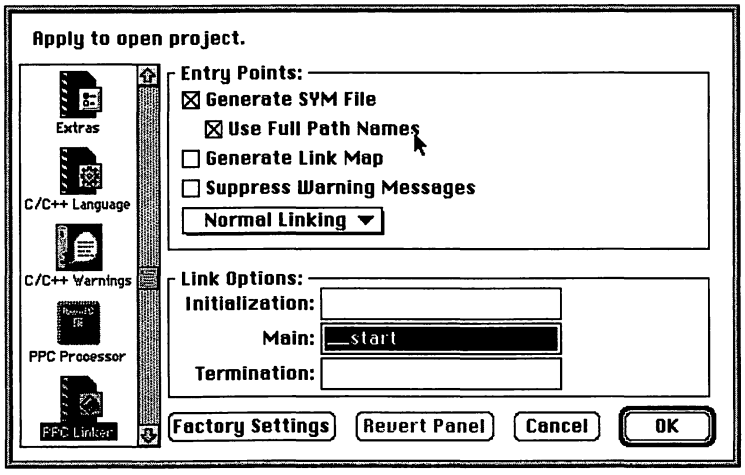


Figure 6.2 Setting the linker to produce symbols for the debugger.

Important

Before you start the CodeWarrior source debugger, check that you have installed the auxiliary files it requires to operate. All 680x0 and Power Macs running System 7.5 or later require the Extension file "MetroNub" for low-level support. Power Macs must also have "ObjectSupportLib" in the Extension folder. For a Power Mac running System 7.1.2, the "PPCTraceEnabler" file must also be located in this folder. If these files aren't present, look for them in the System Extensions folder on the CodeWarrior CD. Copy these files to the pertinent System Folder directories on your target system and reboot it.

To launch CodeWarrior's high-level debugger, double-click on the MW Debug application, or drag the project's .xSYM file icon onto the MW Debug icon. MW Debug launches, and after a brief interval, two windows appear (see Figure 6.3) The front window, titled SwitchBank, is the Program Window. It displays the source code file that has the active function (in this case,

`main()`). The other window, titled SwitchBank.xSYM, is the Browser window. It's used to select other source files in the project, so that you can examine them and set up breakpoints. The floating Toolbar provides ready access to often-used items in the Control menu. If you're more comfortable using the keyboard to step through a debugger, you can get rid of the Toolbar by clicking on its Close box.

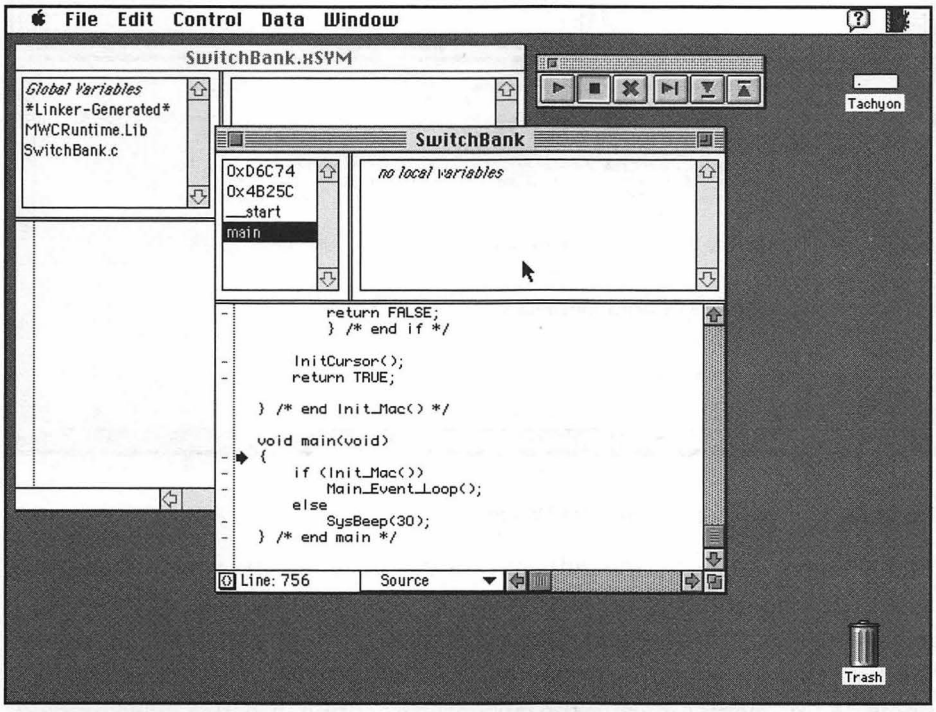


Figure 6.3 MW Debug displaying the Program and Browser windows.

Let's take a closer look at the Program Window (see Figure 6.4). It's composed of three panes, or sections. The bottom section is the Source Pane. It shows the source code of the active function. It's where you step through your program, one line at a time. Tick marks on the pane's left indicate executable statements. The small arrow adjacent to these marks points to the currently executing statement.

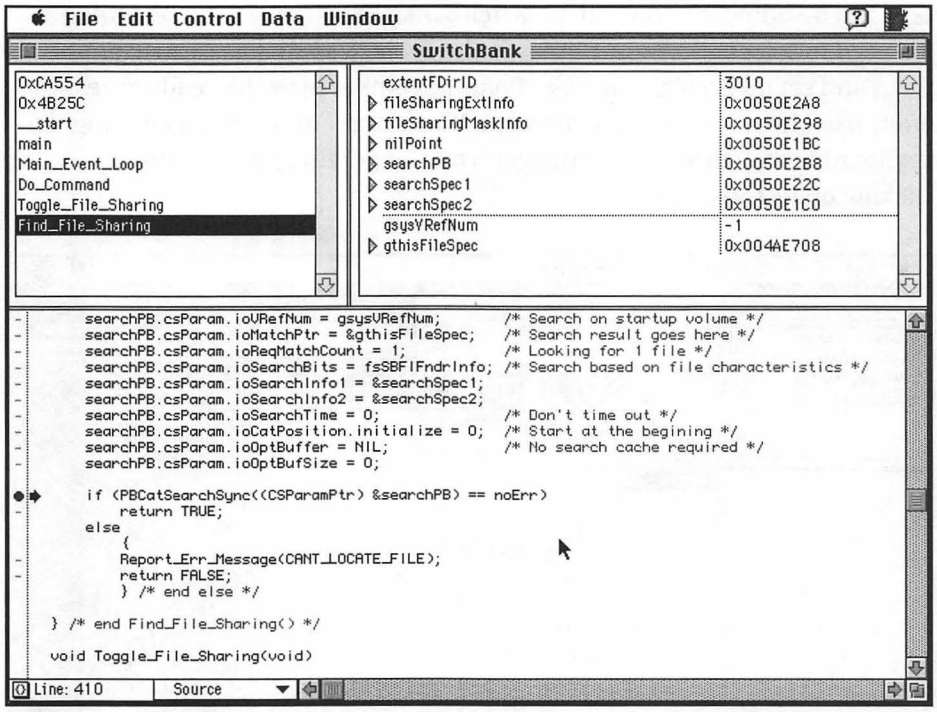


Figure 6.4 Details of the Program Window.

At the Source Pane's bottom left are various indicators and controls. Starting at the left is an icon with the character "T." This indicator manages thread debugging (680x0 Macs only). When you click it, a popup menu displays any program threads operating under the Thread Manager. An underlined thread name on this menu indicates the currently executing thread. A checkmarked thread name indicates that the Program window shows the source and variables for this thread. In other words, you can examine that thread's variables, call chain, and source code even though it's not currently executing.

The small braces, or Function, icon operates like its counterpart in the CodeWarrior IDE's editor window. When you click on it, a popup menu appears that displays all of the functions in this file. The checkmark in this menu flags the active function. If you select another function, the Source Pane displays the source code of that function, starting at its entry point. To the right of the Function icon is an indicator that shows, for this file, the

source line number of the currently executing statement. Finally, next to the line number indicator there's a popup menu that lets you change the Source Pane's display from C source code to the corresponding assembly language statements generated by the compiler (see Figure 6.5). You can single-step through 680x0 or PowerPC assembly language code, and set breakpoints if you choose to do so.

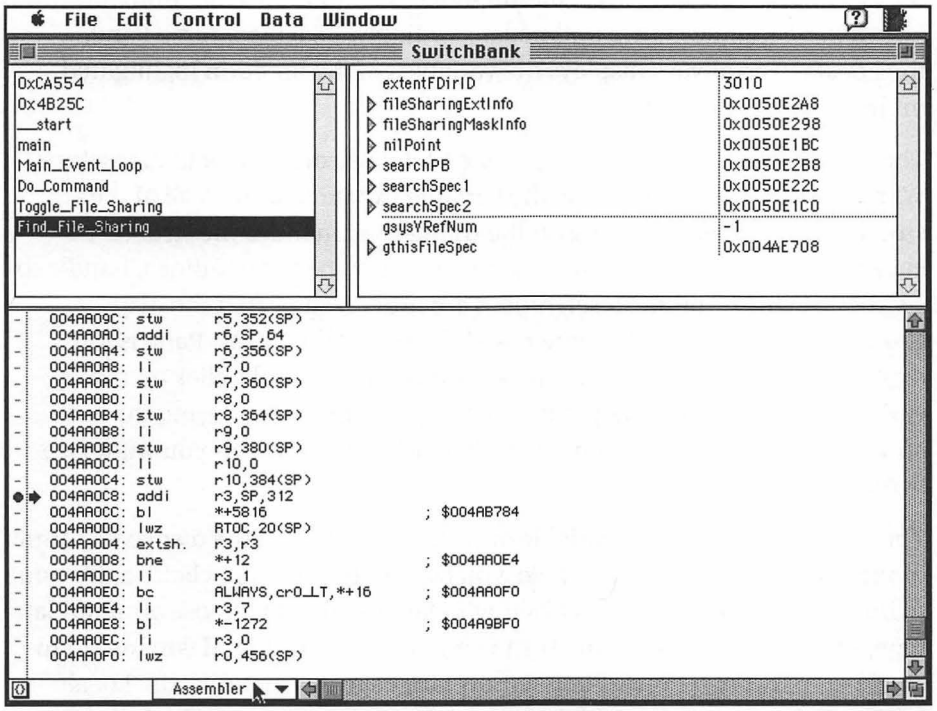


Figure 6.5 Viewing the program's code as PowerPC assembly language.

The pane in the Program Window's upper left is termed the Call Chain Pane. It displays the list of functions called prior to the function shown in the Source Pane. In Figure 6.5, the highlighted name, `Find_File_Sharing`, is the active function. From the list, you can see the `main()` called `Main_Event_Loop()`, `Do_Command()`, and `Toggle_File_Sharing()` before calling `Find_File_Sharing()`. The Source Pane's output is tied to the highlighted choice in the Call Chain Pane. Clicking on another function name in the Call Chain Pane highlights that name and immediately takes you to that function in the Source Pane. The

Source Pane displays this function's source code at the point where it called the next function in the chain.

The pane in the upper right portion of the Program Window is the Locals Pane. It lists the function's local variables, plus any static or global variables referenced by the function. A dashed line separates the function's local variables (at the top of the pane) from the global variables (at the pane's bottom). When the flow of execution moves to a different function, the Locals Pane's contents are updated accordingly. Like the Source Pane, the Locals Pane always displays the variables of the function highlighted in the Call Chain Pane.

The small triangle to the variable name's left indicates that it is a structure. When you click on the triangle, the variable expands to show all of the structure's elements. Clicking on the triangle again hides the structure's elements. When you hold down the Option key when expanding a handle to a structure, the multiple dereferences are processed so that the display shows the structure's data elements. If the size of the Locals Pane is too confining, especially for large data structures, just double-click on the variable name. A new, independent window appears, displaying the entire structure. You can create as many independent windows as you want (see Figure 6.6).

The current value of each variable appears to its right. If the displayed format of the variable's data is unsuitable, you can change it. First, click on the value to highlight it. Then, go to MW Debug's Data menu and choose another data type, say, character. The value is shown in the new format. If you intend to single-step through PowerPC assembly language instructions, the Locals Pane still displays the variable's contents as you continue through the program.

You can edit the contents of certain variables by double-clicking on the value. The data becomes *framed* (surrounded by a box). This indicates that you can enter a new value. The types of data you can enter are decimal (signed and unsigned), hexadecimal, floating-point, fixed-point, characters, and strings (C or Pascal style). Character data must be enclosed in single quotes; Pascal strings must be enclosed in double quotes and include the `"\p"` escape sequence. The values of pointers to data structures cannot be edited.

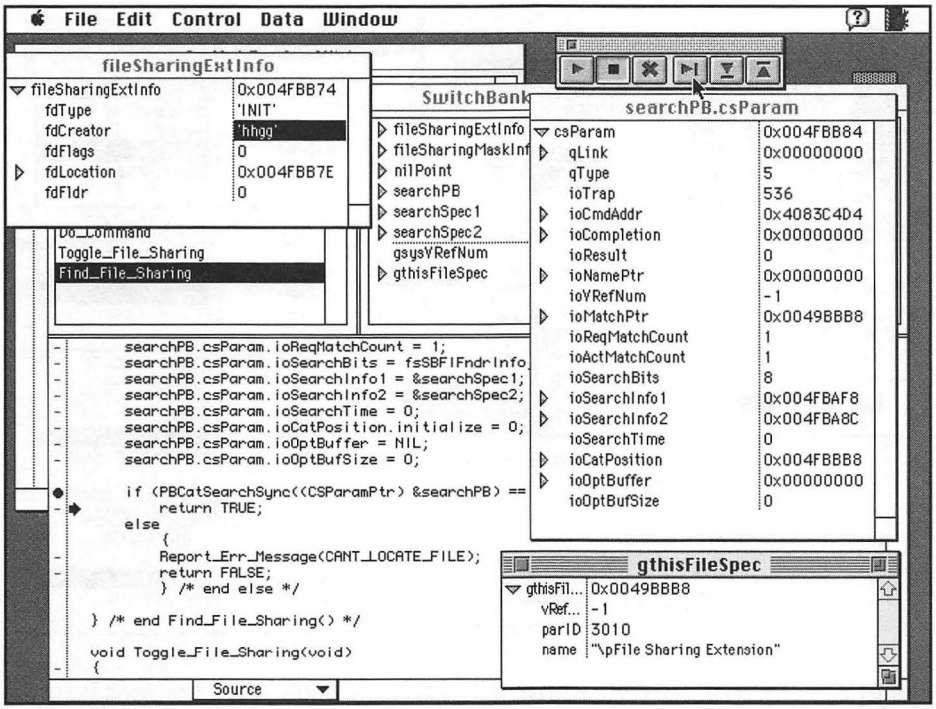


Figure 6.6 Displaying data structures in their own windows.

The Browser Window operates basically like the Program Window. However, while the Program Window is focused on the active function in a file, the Browser Window is oriented toward dealing with the program's files and variables as a whole. The bottom area is the Source Pane, and it displays the contents of the selected file. The upper part of this window has three panes, similar to a class browser. The left pane is the File Pane, which displays the names of the files used to produce the application. The middle pane displays all of the functions in the chosen file. The upper right pane is the Globals Pane, and displays the global and static variables that are shared across all of the files (see Figure 6.7). Notice that for array `AEInstalls[]`, you get a special window where you can alter the array's size and bind it to an address, a variable, or a register.

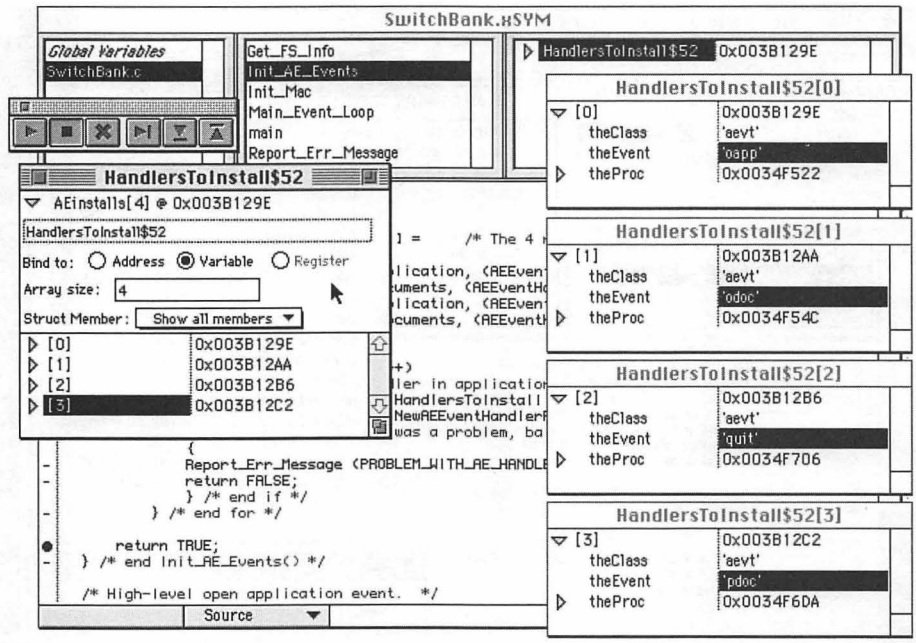


Figure 6.7 The Browser Window, with a display of the application's high-level event dispatch table.

Now it's time to control the program. You do this using commands from the Control menu. The Run, Stop, and Kill commands provide gross program control. The Run command simply starts the program from `main()`, or the location where the program was halted. The Stop command suspends program execution. If you issue another Run command, the program resumes execution from this point. The Kill command terminates the program under test. If you issue a Run command after a Kill command, the program's execution starts over in `main()`.

The Step Over, Step Into, and Step Out commands apply fine control over the test program. The Step Over command executes program code, a line at a time. You'll use the Step Over command frequently to single-step through a program, observing the results of Toolbox calls and tracking the course of C control statements. If the current line of code is a function or Toolbox trap, MW Debug calls the function call or trap, returns, and advances to the next

source line. In this sense, the debugger appears to "step over" function calls, even though their code actually executes. The Step Into command, when invoked, carries you inside the function called by the current source line. This allows you to examine what values get passed to the target function, and examine the operation of that function's code. Note: You can Step Into the code of libraries or other files that don't supply a symbols file, but you can only view the trace as assembly language in the Source Pane. The Step Out command executes the remaining code in the current function and halts the program once it returns to the calling function.

The VCR-style button icons on the Toolbar correspond to the gross and fine program control commands on the Control menu. The three icons that make up the Toolbar's left correspond to the Run, Stop, and Kill commands, while the trio of icons on the right represent Step Over, Step In, and Step Out.

To mark or place a breakpoint in a program, use the Function icon and popup the function list to jump to a suspect function. Next, scroll through the source code to the questionable statement. Place the pointer on the statement's tick mark (it's located on the left side of the Source Pane) and click on it. A circle appears, replacing the tick mark. The circle indicates that a breakpoint is set for this statement (see Figure 6.8). You can set as many breakpoints as you like. To remove a breakpoint, click on the circle again. To remove all of the breakpoints at once, choose Clear All Breakpoints from the Control menu.

If you're in the middle of a marathon debugging session, and wonder what breakpoints you've set, you can display all of them by selecting the Breakpoints item in MW Debug's Window menu. This window displays every breakpoint and includes information on the function name where each breakpoint is set, and the corresponding line number of the source code statement in the file. A double-click on a breakpoint takes you to the Browser Window, which displays the source code statement that has the breakpoint.

You can also monitor critical sections of memory using watchpoints, if your Mac is running System 7.5 or later. (680x0 Macs must also have virtual memory turned on.) To do this, halt the program in the suspect area by use of a breakpoint. Now, select a target variable in the Locals pane by clicking on it, and pick Set Watchpoint from MW Debug's Data menu. (You'll use a different method to select a range of memory.) Start the program.

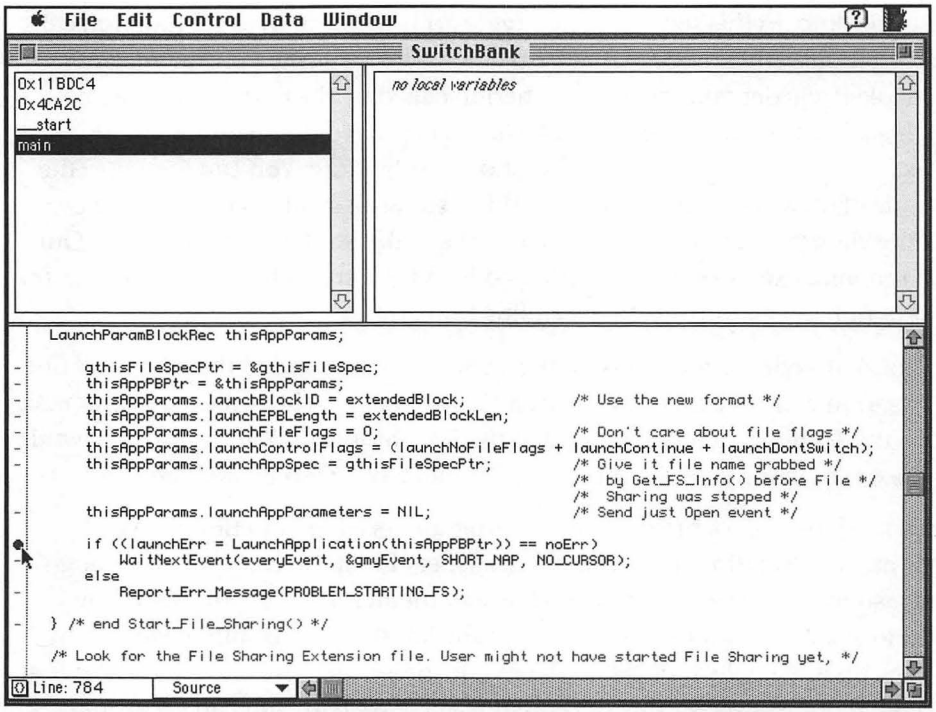


Figure 6.8 Setting a breakpoint.

When the variable changes, you get an alert, and the program halts. Now you can snoop around, observing where the program stopped, and check the values of other variables. Like breakpoints, a Watchpoint window display what watchpoints you've set. Note that watchpoints work only with globals, variables, or objects allocated on the application's heap. You cannot set watchpoints on register-based and stack-based variables, or low-memory globals.

Important

Under System 7.1.2, MW Debug uses the "PPCTraceEnabler" and "MetroNub" Extensions to access the Power Mac's debugging facilities. MetroNub provides access to low-level debugging services that set/reset breakpoints, kill processes, and perform memory reads and writes. MW Debug uses low-level message blocks (via the PPCToolbox) to communicate

with these services. These services in turn control the test application, and return information about its behavior back to MW Debug.

PPCTraceEnabler is a native Extension written by Apple. It gives MW Debug access to the single-step trace bit in the PowerPC processor's Machine State Register (MSR). The MSR is a supervisor-level register, and is not normally accessible by user-level application code. The PPCTraceEnabler Extension sets up this access. On Power Macs running System 7.5 or later, all that's required is the MetroNub file. On 680x0 Macs, MetroNub patches the appropriate trap entries in the dispatch table to provide low-level debugging services to MW Debug.

For your application to be controlled properly by MW Debug, it must have the `canBackGround` bit set in the `SIZE` resource, and the program must make frequent calls to `WaitNextEvent()`. For more information on the `SIZE` resource, see Chapters 3 and 5.

MW Debug remembers the size and location of the Program and Browser Windows, and the locations of all the breakpoints. This information is stored in a file with an extension of .dbg. Continuing with the earlier example, if you debug the application Klepto and set some breakpoints, MW Debug produces a file named "Klepto.dbg." If you want to quit MW Debug, and resume the job later with all the breakpoints in place, don't delete the .dbg file.

Because you've got SwitchBank up and running under MW Debug, let's do a short tour. In the Program Window, click on the Function icon and select `Init_Mac()` from the popup menu. Scroll through `Init_Mac()`'s code and set a breakpoint on the statement containing the Apple Event initialization function, `Init_AE_Events()`, as shown in Figure 6.9. Now pick Run from the Control menu, or type ⌘-R. After a short delay, SwitchBank should halt in `Init_Mac()`, at the call to `Init_AE_Events()`. Note: If you hold down the Option key when you set a breakpoint marker, the program automatically executes to that breakpoint, unless it encounters another breakpoint.

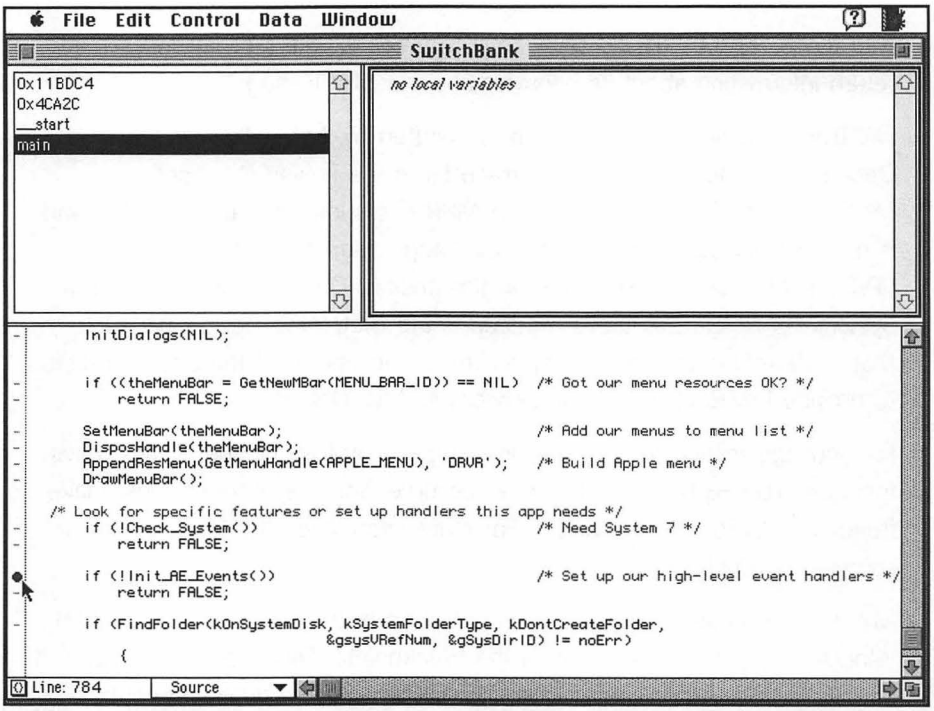


Figure 6.9 Setting a breakpoint for the Apple Event initialization function.

If you wanted to step over this function, you would simply pick Step Over in the Control menu or type ⌘-S. (After single-stepping through lots of code, you'll soon appreciate this command's keyboard equivalent.) However, let's examine how this function operates. Type ⌘-T to step into `Init_AE_Events()`. The Source Pane now displays this function's entry point (see Figure 6.10).

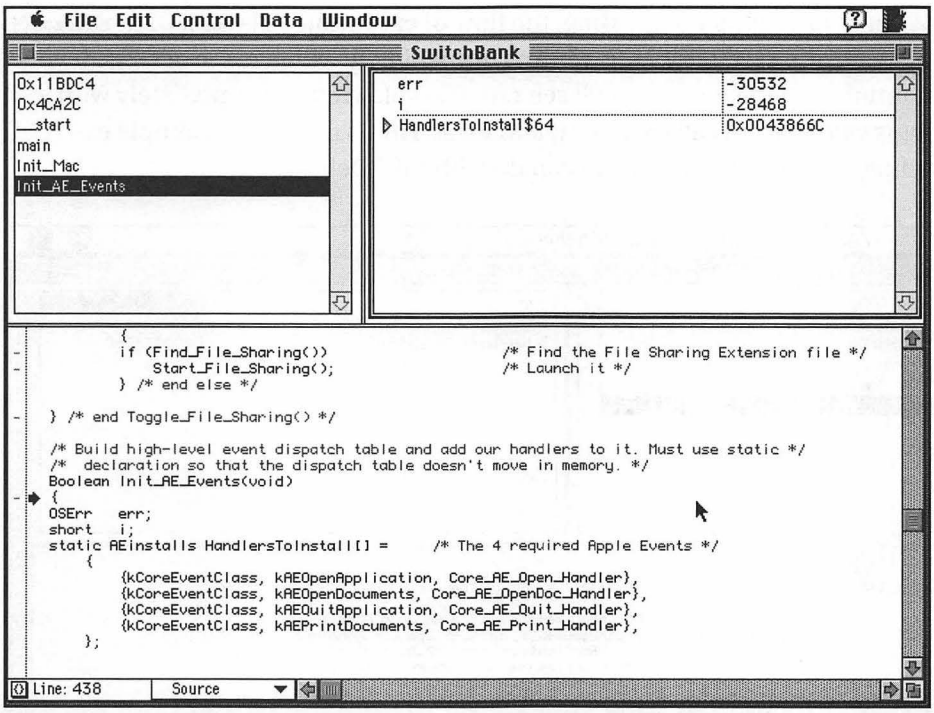


Figure 6.10 Inside the Apple Event initialization function.

Type ⌘-S several times and watch the variable `i` get initialized by the `for` loop. Single-step and observe the result passed back by `AEInstallEventHandler()`, and how the `if` statement within the loop checks for an error result. Step through the loop once or twice, and when the execution pointer arrives back at the `if` statement again, go to the Locals Pane and type a negative number in `err`'s value area to simulate an error (see Figure 6.11).

When you single step this time, the flow of execution calls `Report_Err_Message()` instead, and `Init_AE_Events()` returns immediately with a value of `FALSE`. If you continue to single-step, you'll see `Init_Mac()` also return immediately with a `FALSE` value, `main()` calls `SysBeep()`, and exits. This is obviously a simple example, but it shows what you can do with MW Debug.

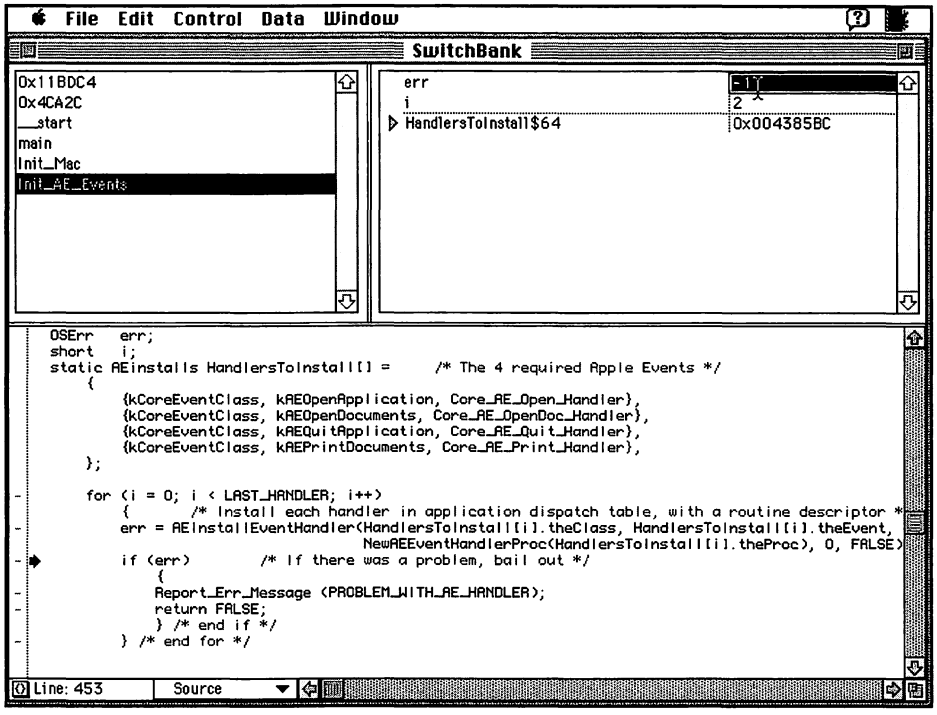


Figure 6.11 Changing the value of `err`.

Let's look at another section of SwitchBank. If you had let SwitchBank terminate from the last example, the Source Pane in MW Debug's Program Window states that SwitchBank is not running. Go to the Control menu and choose Clear All Breakpoints. Type ⌘-R to run SwitchBank again. After a brief delay, the Program Window's panes should fill with source code. The Source Pane should be positioned in `main()`, ready to go. Click on the Function icon, popup the function list menu, and pick `Core_AE_OpenDoc_Handler()`. You'll wind up at the entry point to this function, as shown in Figure 6.12. Now set a breakpoint at the first executable statement in the function.

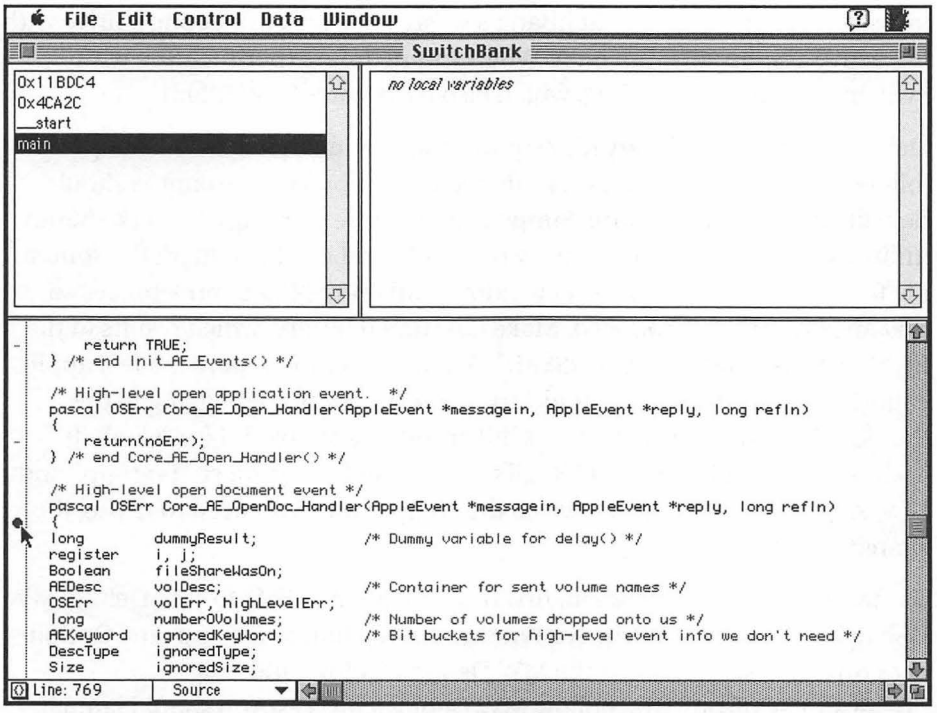


Figure 6.12 Adding a breakpoint to the high-level Open Document function.

Once that's done, type ⌘-R to run SwitchBank. The panes in the Program Window should clear, and the Source Pane contains a message stating that SwitchBank is executing. So far, so good. Now you need to create a high-level Open Document Apple Event. Go to the edge of Program Window, hold down the Option key, and click on the desktop pattern. MW Debug's two windows should disappear, leaving a clear view of the Macintosh desktop. If you pull down the Application menu, you can see that MW Debug and SwitchBank are running, but not visible. The SwitchBank icon doesn't appear to be active (it doesn't display the "hollow" icon that active applications use), but that's because MW Debug had the Process Manager launch SwitchBank behind the Finder's back. Not knowing this, the Finder hasn't updated SwitchBank's Desktop icon to reflect this fact. Click on the startup volume's icon, drag it to the SwitchBank folder, and drop it onto SwitchBank. MW Debug should reappear, with SwitchBank's execution suspended inside the `Core_AE_OpenDoc_Handler()` function. You can single-step through this function, and observe how information is obtained from the Apple Event

message. You'll also see SwitchBank's safety logic balk at ejecting a drive with the active system software on it. When you're done experimenting with SwitchBank, quit MW Debug, which also terminates SwitchBank.

Debugging a shared library file requires that you open it in MW Debug first, followed by the application that's linked to the library. An example should help illustrate the procedure. Suppose that you're working on a set of handy utility functions in a shared library named "CoolLib." First, mark the source file for debugging in the Project window, and set the linker preferences so that an .xSYM file is produced. Make the shared library, which results in the files "CoolLib" and "CoolLib.xSYM." You also have to prepare the test application that gets linked to your library. Mark the file for debugging in the Project Window and use the same linker settings as you did for "CoolLib." Make the test application. Let's call the output of this project "TestApp" and "TestApp.xSYM." Now you have all the components you need to debug the shared library.

To start the debugging session, first drag "CoolLib.xSYM" to MW Debug. MW Debug launches, and a Browser Window for the library file appears. Option-click on the desktop to hide the MW Debug window, and drag "TestApp.xSYM" onto the hollow MW Debug icon. TestApp should launch, and you have three windows on-screen: the Browser Window for "CoolLib.xSYM," the Browser Window for "TestApp.xSYM," and the Program Window for TestApp (see Figure 6.13). Next, you set breakpoints in the shared library code using the CoolLib.xSYM window. To reach the breakpoint so that you can begin code tracing, start TestApp in the Program Window by selecting Run from the Control menu.

MW Debug can debug certain types of code resources, such as Photoshop plug-in modules and HyperCard XCMDs. To debug a code resource that's been copied and pasted into an application, first rename its .xSYM file using a bogus name. Drag and drop this file on MW Debug. When MW Debug launches, it uses the name of the symbols file to find the proper executable file. When MW Debug can't locate your fictitious executable file, it prompts you for its location using the Standard File dialog box. Navigate to the application hosting the code resource, click on Open, and begin your debugging.

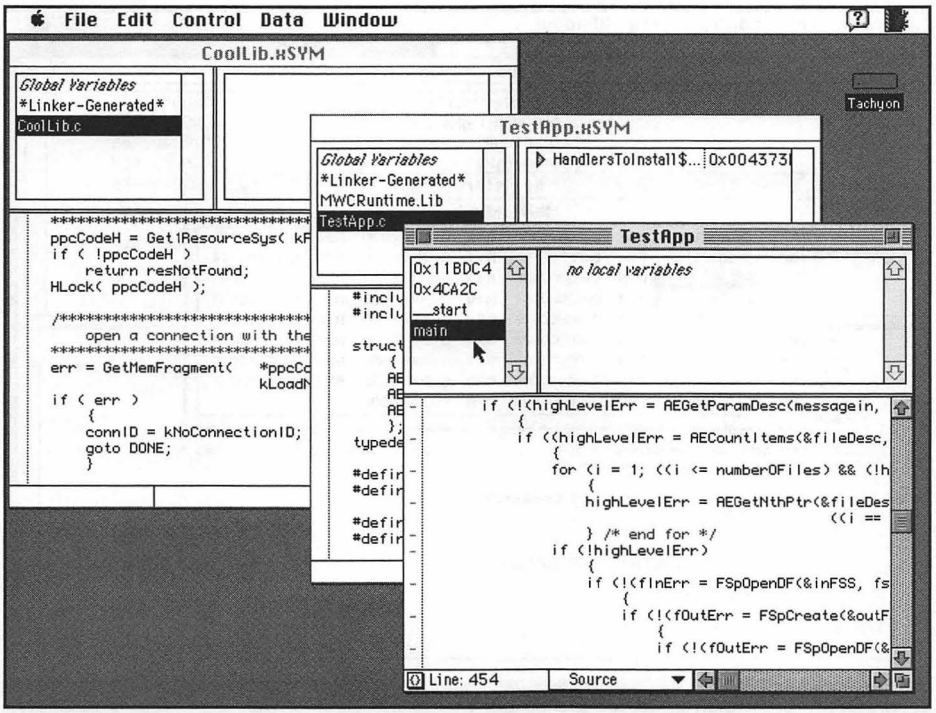


Figure 6.13 Debugging a shared library.

MW Debug also offers some surprisingly low-level debugger features. These are available whether you're debugging an application, a shared library, or code resource. If you select the Show Registers item under the Windows menu, a window displaying the processor registers for either the 680x0 or PowerPC processor appears. Better still, by double-clicking on a register value, the value is framed so that you can modify the register's contents (see Figure 6.14). Because you're messing with the state of the processor itself, use this capability with extreme caution. Another menu item, Show FPU Registers, lets you examine the floating-point registers of the 68040 or PowerPC.

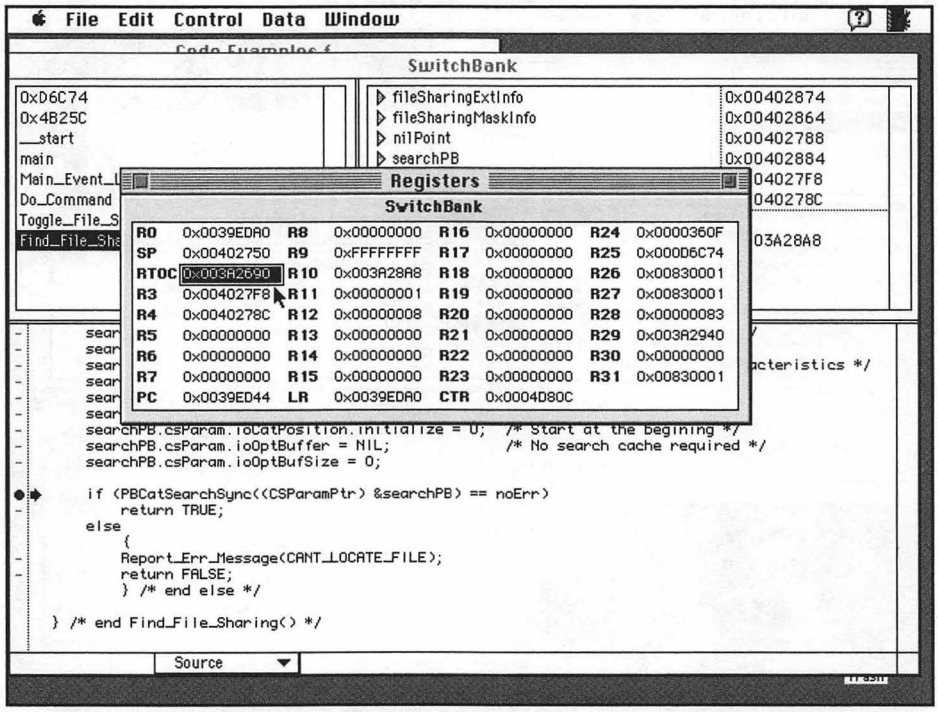


Figure 6.14 Changing the TOC register on the PowerPC processor in MW Debug.

You can view the contents of raw memory by using the View Memory command from the Data Menu. A window appears, displaying the contents of RAM starting at the default address of 0x0. You can use the View Memory as... item in this menu to display the memory in a form convenient for you to interpret. For example, select View Memory as..., and click on the `'char'` data type in the dialog box. Now, choose View Memory. A window labeled Memory 1 appears. Go to the address slot at the top of this window, and type **`0x910`**. You should see the Pascal string `"\pSwitchBank"` as the current application. (Remember from the walkthrough of the `ShowInitIcon()` code that 0x910 is the low-memory global that stores the currently running application.)

Low-Level Debuggers

For PowerPC program debugging using a single Power Mac, two low-level debuggers are available: Apple's MacsBug 6.5.2 and Jasik Design's The Debugger.

MacsBug

MacsBug is a combination of 680x0 and PowerPC code, and can thus debug either 680x0- or PowerPC-based Macs. This capability makes its installation simple—you just drag it to the System Folder, and reboot. MacsBug isn't an Extension file: During the early stages of the boot process, the Mac OS looks for a file by this name in the System Folder. If the file is present, the bootstrap code loads MacsBug's code into memory. MacsBug then installs its own exception handlers to field system errors and to implement breakpoint and single-stepping capabilities. Finally, it patches entries in the A-trap table. If everything goes smoothly, you'll see a message "Debugger installed" on the "Welcome to Macintosh" splash screen.

The initial 6.5 release didn't offer much PowerPC support, but version 6.5.2 has been updated to let you examine PowerPC code and the state of the PowerPC processor registers, even after the worst of system crashes. MacsBug is a lean and mean low-level debugger, and uses only about 150K of RAM. This sparing use of system resources makes MacsBug virtually crash-proof, and lets you debug tricky code resources such as MDEFs, INITs, interrupt handlers, and drivers. Unfortunately, such minimalist use of the system forces MacsBug to offer a horrendous character-only display. You also have to remember what commands to type into its command line. Until you become familiar with MacsBug's commands, keep its manual nearby.

MacsBug lets you single-step through PowerPC code, and it handles any Mixed Mode context switches when the thread of execution hops to a 680x0 Toolbox trap or plug-in module. This is handy for situations when your program calls `CallOSTrapUniversalProc()` or `CallUniversalProc()` to hop into a function with a different ISA. You don't see any Mixed Mode Manager code or switch frames; instead, the debugger halts in the module's first 680x0 instruction, and MacsBug automatically displays emulated 680x0 processor registers and 680x0 disassembly code. Finally, you can disassemble code fragments into their respective PowerPC assembly language statements.

Important

All too often while testing new code, the Mac simply seizes and hangs. Sometimes, the program hangs in an infinite loop. More often, critical low-memory globals that maintain the operating system get clobbered. You can occasionally recover by issuing a non-maskable hardware interrupt (NMI), which activates a low-level debugger. You use the debugger to assess the damage, and then kill the buggy application process. If the Mac OS is in reasonable health, the debugger will eventually exit and you'll be back in the Finder. You should immediately close any other applications. (It's not a good idea to run other applications during your development work, unless you need them to test AppleEvent communications or an OpenDoc part.) Now pick Restart from the Finder's Special menu. This sequence saves critical data, updates the hard drive's file directories, and lets you resume debugging with a clean system.

If the Mac OS is badly cooked so that the NMI has no effect, you have to resort to a hardware reset. This forces the processor through a reset, which in turn reboots the computer. In the old days, Macs had a programmer's switch with two buttons on it. One button triggers the hardware interrupt; the other is for the hardware reset. Today's Power Macs lack a programmer's switch, but with the appropriate keystrokes, you can initiate an NMI or a reset:

Key Sequence	*Result*	*Purpose*
⌘-Power Key	Interrupt	Starts low-level debugger.
⌘-Control-Power Key	Reset	Forces hardware reset, restarts processor and system.

Of course, you don't have to wait for the Mac to crash to invoke MacsBug. The best way to use MacsBug is to launch the errant application, hit the interrupt sequence (⌘-Power Key) to summon MacsBug, and start exploring the application's partition. It's a good idea to type **HZ** (Heap Zone), press Return, and look at a list of memory partitions for the various processes and the system heap. If your program seems to be striking out at random,

clobbering itself or other applications, you might want to jot down the addresses of some of these other zones.

When you're ready for a walk on the wild side, type G (for go) and press Return. The application takes control, and resumes execution from the current PC (program counter). The next step is to run the program through the event sequence that triggers the crash; when the crash occurs you should see MacsBug.

Afterward, the first thing you need to do is check that you're still in your application. Look in the status region of the MacsBug screen (it's sandwiched between the stack display and the register display at the screen's left) to check for the current application name. If the name isn't your program, an errant access by your program may have trashed another application. This application in turn crashed when it got processor time.

To figure out what happened, the first thing to check is instructions located around the crash site. To do that, type IP and press Return; this command displays a half page (approximately 64 bytes) of disassembled machine code centered around the address in the PC. You can also type an HD (heap display); MacsBug will show the status of blocks in memory, such as master pointers, and various resources. You can display memory by typing DM (display memory) and an address. Type SC and Return to get a stack crawl. This command displays a section of memory starting at the top of the stack, where you can examine the chain of functions that were called just before the crash. If another application has crashed, and you want to spelunk down into your application's heap, type HX (Heap Exchange), followed by the starting address of your program's partition (which you obtained using the HZ command). Press Return, and this will display the memory zone your program occupies.

After you examine the crash site, you might want to kill the program. Normally, typing ES (Exit to Shell) kills the process, backs out of MacsBug, and gives you access to the Finder. If that doesn't work, try typing RS, for restart. This command unmounts all of the volumes or drives, and restarts the computer. In tough cases, you will need to type RB for reboot. This command is nearly equivalent to the hardware reset in that the system reboots unconditionally. However, where it differs is that the RB command first attempts to unmount the startup drive in an effort to minimize damage to its file directories.

Although it is not complete or exhaustive in detail, Table 6.1 includes the most useful MacsBug commands. For additional information check out the *MacsBug Ref & Debugging Guide*, published by Apple.

Table 6.1 MacsBug's Most Useful Commands

Command	*Action*	*Description*
ATB *trap*	A-trap break	Stops program execution on the specified trap.
BR *addr*	Breakpoint	Sets breakpoint at address. If no address is specified, displays list of breakpoints.
BRC *addr*	Breakpoint clear	Clears the breakpoint at the address. If no address is specified, all breakpoints are cleared.
CS [[*addr*] *addr*]	Checksum	Checksums memory at the address. If two addresses are specified, the memory between these addresses is checksummed.
DM [[*addr*] *nbytes*]	Display, Memory	Displays *n*bytes of memory starting at the address specified. Default is 16 bytes.
EA	Exit to application	Kills the current application and restarts it.
ES	Exit to shell	Kills the current application and exits to the Finder.
G *addr*	Go	Resumes program execution from the specified address. If an address is absent, execution resumes using the current PC.
GTP *addr*	Go until PowerPC code	Resumes program from specified address until PowerPC code is reached.

Command	*Action*	*Description*
HC	Heap Check	After a crash, use this command to validate the integrity of the current heap.
HD	Heap Display	Displays memory blocks in the current heap.
HELP command	Help	Provides a brief explanation of the command.
HS	Heap Scramble	Relocates heap blocks after every A-trap call. Useful for uncovering memory problems.
HX *addr*	Heap Exchange	Changes the current heap to the one specified by the address.
IL [*addr*] *n*]	Disassemble from address	Disassembles (680x0 or PowerPC) machine codes, starting at address. Default address is current PC. The *n* is the number of lines to display.
ILP [*addr*] *n*]	Disassemble PowerPC code	Disassembles PowerPC code starting at address for *n* lines.
IP *addr*	Disassemble around address	Disassembles (680x0 or PowerPC) machine codes around the address. Default address is the current PC.
RB	Reboot	Unmounts all volumes and restarts system.
RS	Restart	Unmounts hard drive, forces system reset.
S [*n* | *expression*]	Step	Single-steps through code *n* instructions, or until condition determined by expression is met.

continues

Table 6.1 Continued

Command	*Action*	*Description*
SC6 [*addr*] *nbytes*]	Stack crawl	Tracks usage of A6, the frame pointer (680x0 processor). Crawl starts at specified address for *n*bytes.
SC [*addr*] *nbytes*]	Stack crawl	Tracks usage of A7 (680x0) or r1 (PowerPC) the stack pointer. Crawl starts at specified address for *n*bytes.
SM *addr value*	Set memory	Sets memory at the specified address to value.

The Debugger

There's another low-level debugger available that also gives you access to the PowerPC registers, displays native instructions, and lets you debug a native program. It's appropriately named The Debugger and is written by Steve Jasik of Jasik Designs. Because it's a low-level debugger, The Debugger can be used to debug stand-alone code such as Extensions, MDEFs, and CDEFs. For a low-level debugger, it sports some sophisticated high-level debugger features, including windows and menus. Because of the high-level interface, you don't have to remember any commands: you just point and shoot from the menus.

Background Info

How does The Debugger provide a high-level system interface while maintaining a low-level debugger's robustness and capabilities? The Debugger copies the required system resources into a private area owned by it. This enables The Debugger to provide high-level debugger services, yet still continue to function when the operating system gets mangled by a program bug. It performs this sleight of hand at boot time using a support file called "MacsBug." The MacsBug name fools the Mac OS into loading

a program that first allocates a block of memory, and then copies the system code resources into this area. A second Extension file, "xDbgrStartup," loads The Debugger into this private memory area and starts it.

The Debugger requires a number of other support files. One such file is a specially modified version of "PPCTraceEnabler." Another file, "ROM.snt," contains information about the Power Mac ROMs that The Debugger uses to set up the private area. It's also used to help The Debugger track patches made by operating system code and third-party Extension files.

Another critical support file is MacNosy II, which is a 680x0 and PowerPC disassembler program. MacNosy is how Steve first established a reputation in the Mac community, and it's a vital component of The Debugger. Because of fundamental changes in the second-generation Power Macs' hardware (the PCI bus, and the Open Transport network interface), they have radically different ROMs from the original Power Macs. This in turn creates a problem for The Debugger because the standard Power Mac ROM.snt file was out of sync with these new ROMs.

Fortunately, because MacNosy is bundled with The Debugger, you can use it to create an up-to-date ROM.snt file. Power Mac 9500 owners in June of 1995 had to manually create the ROM.snt file, as did 7200/7500/8500 owners in August 1995. However, with the October 1995 release of The Debugger, Steve remedied this problem by making the ROM.snt update automatic. Here's how it works. At boot time, the startup files just described determine whether the current ROM.snt file matches the host system's ROMs. If not, once the Finder loads, it's commanded to launch MacNosy II. MacNosy II scans the ROMs and installed patch code, and then uses this information to build a new ROM.snt file. After this file is made, MacNosy II quits, completing the setup. This enables The Debugger to be used immediately with the latest Power Mac systems released from Apple or clone vendors without a major configuration hassle.

When a Mac boots with The Debugger installed, you'll see a "Debugger installed" banner, just like when MacsBug loads. You then see various icons

appear as the Extensions install. After the xDbgrStartup Extension loads, a dialog box pops up, in which you can adjust various debugger settings, such as the amount of memory to allocate for The Debugger, and which screen it uses on a multiple-monitor system.

When you click on the Launch "The Debugger" button or press Return, the screen displays lots of windows temporarily as The Debugger sets up housekeeping. The Control Panels then load, followed by the Finder, and the boot process completes. Like MacsBug, you can summon The Debugger by pressing the NMI key sequence (you can also type the alternate key sequence, Option-\). When a system crash occurs, The Debugger appears with a window that displays either 680x0 or PowerPC disassembled code. The disassembly begins with the current PC.

When you launch a 680x0 program, The Debugger automatically reads the .SYM files that CodeWarrior generated for it. (The .SYM file must be in the same folder as the program.) An exception activates The Debugger, and it provides a source code display (see Figure 6.15). The Sigma symbols ( ) to the left of the source code indicate executable statements. If you highlight a statement and type ⌘-B, a small bullet symbol appears that indicates a breakpoint has been set. You can mark either C source statements or 680x0 instructions with breakpoints. By holding down the ⌘ key and clicking on the source window, you can toggle the view between C source code, and C source interspersed with 680x0 machine code instructions. You can single-step through the program as either 680x0 assembly or C source code, viewing the processor registers, and the stack.

On the Power Mac, The Debugger reads the .xSYM file and lets you view a native program as C source code or PowerPC machine code instructions. You can examine the PowerPC processor registers and set breakpoints.
A "training wheels" feature attaches comments beside each instruction to assist you in learning the PowerPC machine code (see Figure 6.16). Like MacsBug, The Debugger automatically handles Mixed Mode switches, and displays the appropriate set of processor registers and disassembled code when the mode switch occurs.

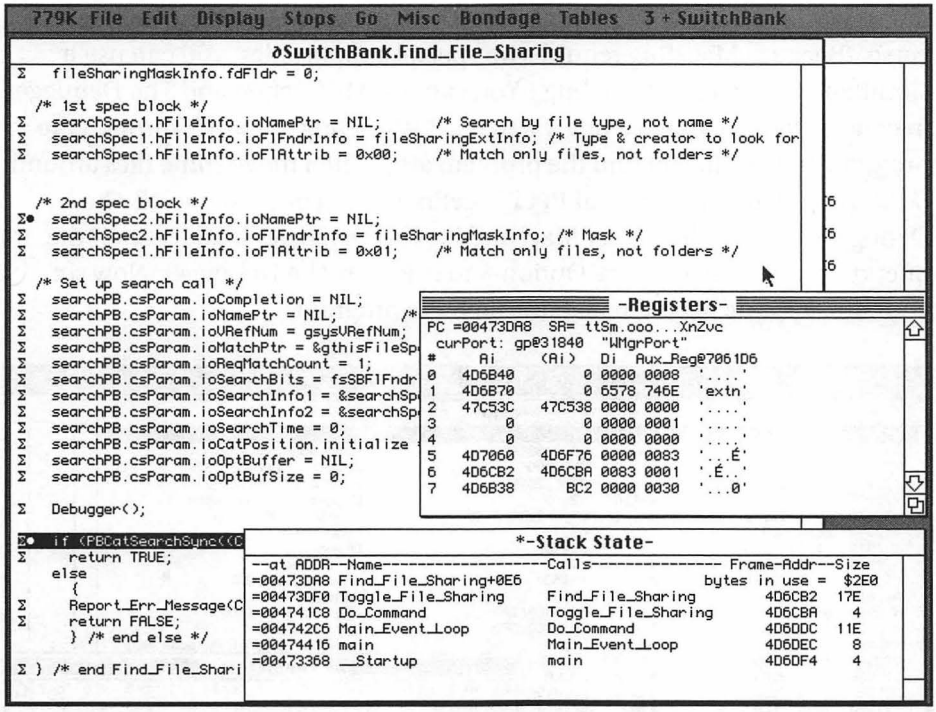


Figure 6.15 The Debugger displaying a 680x0 program as source code; the bullet symbols mark breakpoints.

On either a 680x0 Macintosh or a Power Mac, you can activate a continuous step mode in The Debugger, and watch it run through the program. This is handy when you have to run the program extensively to get to the point in the program where the bug occurs. The stack and register displays are automatically updated during this trace. Pressing ⌘-period stops the continuous trace.

Because The Debugger can use Metrowerks CodeWarrior symbolics files, it is a valuable companion to MW Debug. However, MW Debug uses its own code nub, and The Debugger uses a modified PPCTraceEnabler file. This

means you can't use MW Debug while The Debugger is present, or vice-versa. (Because MacsBug requires no special support files, you can use it simultaneously with MW Debug.) You can use MW Debug and The Debugger in concert to track down a bug. First, use MW Debug to march through the program code until you find the problem area. Then move some files around (MacsBug, xDbgrStartup, and PPCTraceEnabler) in order to install The Debugger. Reboot the Power Mac, and let The Debugger load. Launch the offending program and type Option-\ to bring up The Debugger. Now set your breakpoints or single-step through the program.

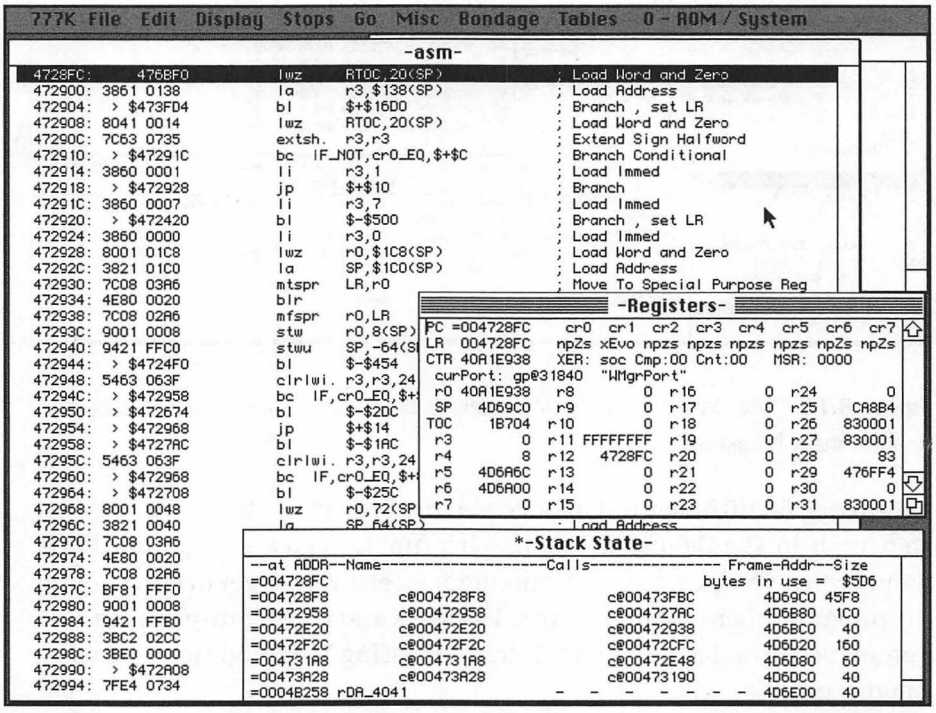


Figure 6.16 The Debugger displaying a PowerPC program as machine code.

One of The Debugger's best features is that you can use it to source-code debug native Extensions. To do this, first move the folder containing your project and source files into the Extensions folder. Next, edit the source code to add a breakpoint statement (`DebugStr()`) inside the Extension's `main()` function. This breakpoint interrupts the boot process so that The Debugger takes control. Open the project, and in the PPC Linker preferences panel, ensure that a symbols file is created (the Generate Sym File item is checked). Go to the PPC Project preferences panel and rename the output file so that the Extension loads after xDbgrStartup. For example, to debug FlipDepth, you name the output file "zFlipDepth." This guarantees that The Debugger loads before your Extension does. Now build the Extension. Continuing with your file example, drag the zFlipDepth Extension and zFlipDepth.xSYM file out of the project folder into the Extension folder. Reboot the system.

When The Debugger's configuration dialog box appears, hold down the ⌘ key and press Return. Keep holding the ⌘ key down until The Debugger appears, with its array of windows. Move the pointer to the -Dbgr Status-window, and double-click on the item named `Dbg_Rsrc`. Its status should switch from OFF to ON. Type ⌘-E to resume the boot process, and several moments later, the breakpoint statement in your Extension's code should activate The Debugger. At this point, The Debugger should be displaying the source code of your Extension. Now you can readily set other breakpoints or watch the values of variables change as you single-step through the Extension.

The best feature that The Debugger provides is a memory-protection mechanism called 'Soft MMU.' As its name implies, this is a software implementation of the PowerPC's memory management unit (MMU). With it, you can assign memory protection for various blocks of memory in your application's partition. Only one PowerPC application at a time can be protected. Access can be set to read-only (block is write-protected), no access (the memory block can't be read or written to), or no protection (the block is accessible to everyone). Once you've assigned the access rights for sections of the program's memory, you switch on Soft MMU by choosing Soft MMU in the Stops Menu and clicking on the OK button in the dialog box that appears.

Because The Debugger's protection mechanism is a software program, the suspect application's execution rate slows by a factor of 30. However, any errant memory access—such as a memory write into a low memory global, or into the application's own PEF container—triggers a warning message. This lets you catch this type of bug immediately, rather than the application inexplicably crashing much later when either the operating system uses a corrupt variable, or the application tours a mangled section of memory.

Future Directions

Copland will use memory protection to safeguard its operating system kernel, device drivers, and Copland-savvy faceless background applications. This prevents wild accesses from an ill-behaved application from crashing the system. However, this doesn't relieve you from tracking down such bugs in your program. For starters, if your application misbehaves and touches the kernel's memory space, Copland responds by killing the process. Simply put, your application goes away, which can be traumatic for the user. Applications also reside in a Compatibility Space, which offers little memory protection among the applications themselves. This arrangement was necessary in order to support certain non-reentrant Toolbox Managers, such as QuickDraw. This means that a bad-mannered application might trash another application. Or, take out the entire Compatibility Space, bringing down all of the applications within it. Blowing up the Compatibility Space requires that your application hammer a precise area of memory used by the non-reentrant Toolbox code, so such an occurrence should be rare. Nevertheless, remember that Murphy's Laws apply, particularly for computers, so take the time to purge memory access bugs from your program.

Debugging Techniques

To the uninitiated, the debugging process might seem arcane, but essentially what it amounts to is gathering information or clues. You observe the program's behavior carefully up to the moment it crashes, taking note of what events trigger the crash. The debugging tools mentioned here serve an

important purpose—they help you prod the remains for additional information, or let you take the program for a tightly controlled stroll over the brink.

This information allows you to determine two things, where the program crashed, and why that particular statement caused the crash. It might seem that all you really need to know is where the program crashed, but sometimes that's not the complete picture. After all, a `for` loop that reads control values out of an array is going to work flawlessly—up until a logic error has the loop read past the end of the array.

There's no exact formula or procedure that you can follow to track down and fix program bugs. Debugging techniques vary on a case-by-case situation because each program is unique. The best technique to minimize program bugs is to code defensively, especially in the user interface code. Remember that the user might perform actions in any sequence. Also, initialize the program with a set of reasonable defaults, because the user might not explore that portion of program where the values of key variables get set. From my own experience writing shareware, it's definitely worth having outside testers try out the program in the early phases. Their efforts invariably point out holes in the interface code. Keep an open mind to the testers' critiques of the program. They often make worthwhile suggestions that can streamline the interface, which in turn results in simpler and more solid interface code.

I mentioned this rule earlier in the book when you wrote your first program, but it's worth repeating: *Always check for errors.* Many Toolbox routines return a value that indicates whether the request completed successfully. In case of a failure, the value returned often indicates what condition caused the error, such as the program ran out of memory, or the disk is full. When using the Resource Manager, you can check the status of calls made to it using the function `ResError()`. Use the `MemError()` function to validate Memory Manager calls. You can obtain the status of some QuickDraw calls using `QDError()`. One of the first functions I add to a new program is `Report_Error()`, so that it can trap any major errors I make when calling unfamiliar Toolbox routines.

I realize it's difficult trying to code for all the possibilities. For example, the apparently simple act of saving a file to disk involves an army of safety checks. You have to see whether a file of the same name exists, ask whether to overwrite the existing file, see if there's enough room on the volume to

save the data, and then constantly monitor the file I/O routines during the save operation. Use MW Debug to modify the results of Toolbox calls so that your error handling code gets thoroughly tested. The benefits from such an effort are a reliable program and robust code that can be reused in future projects.

Of course, if you're doing a quick and dirty in-house program hack, such as the Klepto application, you need not be as exhaustive monitoring the results of the Toolbox routines. Nevertheless, you'll notice that even Klepto performs some safety checks. A lot of Klepto's file setup code came out of the SonOMunger program, and I just replaced the `Munge_File()` function with `Move_Fork()`. This let me knock out a solid and reliable utility program in a short time. However, if you're writing code that you expect other folks to use, do them a favor and do all the safety checking, plus report errors using plain explanations. (Putting an error code in an alert is *not* adequate.) They might not appreciate getting warning or error messages, but users have no patience at all with a program that bombs.

A Bug Taxonomy

There are countless categories and types of bugs. Here I discuss three particular bugs that can be divided into several broad categories. First there are the logic bugs, which are flaws in the algorithms and plague programmers no matter what platform they're working on. The second type is where the Toolbox is called improperly, which either crashes the Mac quickly, or creates hidden damage to the operating system so that trouble rears its ugly head hundreds of instructions later. Finally, there are those bugs that manifest themselves because of side effects that occur in the Mac run-time environment. I will provide some general guidelines for each.

Logic bugs can sometimes be found without resorting to debuggers. A "code walkthrough," where you explain the operation of the program to a coworker or friend, often uncovers gaps or flaws in a program's logic. Also, sit down with a program listing and some paper, and step through the listing line by line, jotting down the values of variables as you manually evaluate each statement. This process, although tedious, can spot some problems. It's also valuable in making you really look at the code, rather than skimming through it with a program editor. Finally, a code walkthrough gives you a pretty good

sense of how the program operates. This operation lets you recognize a problem when some of the variables abruptly acquire new and unexpected values.

During the program's development phase, add code that does limit checks on arrays and other program resources. The overhead of this type of code slows the program, but it will pay for itself when it snares a bug or two while the program is taking form. Limit checks also help in those situations where you're trying to integrate portions of the application that were written by different programmers. Bracket the limit check code with conditional compilation statements so that it can be quickly eliminated during the final build of the shipping application.

Finally, use the compiler to help eliminate logic bugs created by typos, such as the `if` statement that uses a single equal sign in the comparison statement. CodeWarrior's C compilers have a large set of syntax-checking functions you should use to weed out these types of problems. Access the C/C++ Language preference panel and make sure Require Function Prototypes is checked. This setting eliminates the possibility of writing improper Toolbox calls, which is discussed in a moment. Another change you need to make is to access the C/C++ Warnings preferences panel and check most of the items, such as Possible Errors, Unused Variables, Unused Arguments, and Extended Error Checking. These settings force the compiler to watch out for these coding goofs and other syntax problems. Don't hesitate to use MW Debug to step through the code and see what's going on. If you've already done a code walkthrough and have some values you can reference, MW Debug's Locals Pane can uncover algorithmic and syntax bugs quickly.

The next category of bug occurs when you call the Macintosh Toolbox routines improperly. If you're lucky (and usually are), such mistakes take out the Mac fast. You might think that making this sort of goof would be difficult, given the copious documentation on the Toolbox routines. However, such errors do have a way of sneaking up on you.

One type of improper Toolbox usage is calling a routine with arguments that are the wrong size (say, passing a `long` where a `short` was expected). For performance reasons, the Toolbox routines don't carry out any argument checks. On a 680x0 Mac, the stack is used to pass function arguments. If the wrong-sized argument is pushed when making a Toolbox call, it mangles the

stack. The reason for this mess is that when the routine attempts to return, it first pops the expected proper-sized arguments off the stack, even though an improper larger- or smaller-sized argument was pushed onto it by your program. This skews the stack pointer, and consequently the routine fashions a return address out of bogus values sitting on the stack. 680x0-based Macs seize up solid on this type of mistake.

Part of this problem stems from the ambiguity in the size of an integer variable. Depending upon the development tools you use and their settings, an `int` could be 16 or 32 bits in size. My recommendation: Remove `int` from your C vocabulary. Declare variables as `short` or `long` instead.

This type of problem shouldn't occur as often on the Power Macintosh for a number of reasons. As you saw in Chapter 4, Toolbox arguments get stuffed into PowerPC registers rather than pushed on the stack, so a wrong-sized argument isn't as lethal as it would be on the 680x0 architecture, although it's possible to hose the 68LC040 emulator this way. Finally, the ANSI C requirement for Power Mac software reduces this problem because of function prototypes. The header files for all of the Toolbox routines contain function prototypes, and so an argument mismatch in your code is quickly recognized and flagged by the compiler. If you haven't yet checked the Require Function Prototypes item in the C/C++ Language preferences panel, do yourself a favor and set it now.

Another type of Toolbox usage error is where you simply don't supply all the information the routine requires. Guess what happens when that routine uses random data as a source of information? This sort of mistake crops up on those Toolbox routines that use selectors or parameter blocks to pass information.

Background Info

Routines that have selectors operate as follows: they use a single trap word that acts as the entry point into a package of related system services. The selector is a value passed to the routine that determines which function in the package to use. Toolbox routines that fall into this group belong to the Standard File, Alias, Sound, List, Process, Apple Events, Slot, and File Manager.

> Sometimes it appears that a routine doesn't use a parameter block or selector, such as in the case of the Apple Event and File I/O routines you used in the Chapters 3 and 5. However, if you dissect the header files, and pay close attention to what the in-line 680x0 assembly macros do, you'll see that these routines actually use selectors.

For such calls, pay close attention to what arguments the routine requires. I once spent an afternoon trying to figure out why the Slot Manager call `SNextTypeSRsrc()` in my program was reading video sResources from a second display board in a Mac II, rather than the one I wanted. (sResources are special code objects used in expansion board firmware, and are accessed like regular resources, using a name and ID number.) I eventually discovered that I wasn't supplying a value for an argument handling the sResource's ID, called `spID`. So `SNextTypeSRsrc()` looked for the next video resource, indexing off the large nonsense value left in `spID`. With relentless logic, `SNextTypeSRsrc()` dutifully went to the next slot with a video board and found the next sResource for me. Adding a statement to zero `spID` fixed the bug.

Another gotcha lurks in the optional completion functions some Toolbox routines expect. Even if you don't use an I/O completion function with the call to `PBCatSearchSync()` or similar Toolbox routines, place a value of `NIL` in the parameter block to make the fact perfectly clear to the Mac OS. Finally, some routines pass results back to you via a pointer to a buffer you provide. Not to single out `PBCatSearchSync()` here, but you'll recall this routine places the search results in a buffer you allocated for it. Be sure to set up this buffer, or else the routine will hammer at some random memory location with the data you requested.

The last type of bug is what I loosely term "side-effect" bugs. These occur because of side effects induced by certain Toolbox calls or the Mac OS. These bugs are hard to find, because there's nothing obviously wrong with the code. Also, the bug may only bite based upon the application's memory usage and the state of the operating system at certain times. One bug of this type is the memory leak. Certain Toolbox routines create copies of buffers that you're then expected to dispose of. If you don't, eventually the application's memory dries up. As an example of this, look again at the use of the `AEGetParamDesc()` or `GetNewDialog()`. Both allocate buffers that you must

delete when you're finished using them. You might lump this sort of problem under the improper use of the Toolbox, but I make the distinction because you're not actually calling the routines improperly. The program won't crash, but it will eventually run out of memory. This can mislead you as to the real root of the problem. Again, this sort of bug can be avoided by a thorough understanding of what each Toolbox call does.

The other side effect issue is where a Toolbox call or the allocation of a buffer causes the Memory Manager to shuffle things around in memory. For example, if you're accessing a PICT resource (usually to display an image), trouble can occur if the image data gets moved. Here's code that shows how to update a PICT image in a window:

```
/* Globals */
WindowPtr       gBannerWindow;
PicHandle       gThePict;

/Locals */
Rect            thisFrame;
GrafPtr         oldPort;
WindowPtr       whichWindow;

main event loop code...

   case updateEvt:      /* Update the window */
   whichWindow = (WindowPtr) myEvent.message;
   if (whichWindow == gBannerWindow)
         /* It's the banner window */
      {
      BeginUpdate(whichWindow);
/* Start the update */
      GetPort(&oldPort);
      SetPort(whichWindow);
      if (gThePict != NIL)
/* Do you have the image? */
       { /* Display image */
       DrawPicture(gThePict, (*gThePict)->picFrame);
         } /* end if */
      else
         {
         SysBeep(30);
         } /* end else */
```

```
    SetPort(oldPort);   /* Restore port */
    EndUpdate(whichWindow);
  /* Update completed */
    } /* end if == gBannerWindow */
  break;
```

This code works fine as long as the PICT resource `gThePict` stays put. However, if the image data gets relocated, the code that obtains the display rectangle out of `gThePict` (`(*gThePict)->picFrame`) is liable to pass junk masquerading as a `Rect` to `DrawPicture()`. It happens that `DrawPicture()` itself moves memory, so by the time this routine gets to the code `(*gThePict)->picFrame`, the buffer may have moved somewhere else. You'll either get a very weird image onscreen, or a crash. What makes this type of goof deadly is that it occurs only if `gThePict` happens to move. This may or may not happen, depending upon the state of your application's heap at a given moment. You either have to use a handle to extract the rectangle information out of `gThePict`, or lock it in memory, like so:

```
/* Global */
PicHandle      gThePict;

/* Banner intialization code */
Rect           theFrame;
Handle         theLogo;

   (hasColor) ? (theLogo = GetNamedResource('PICT',
                   "\pColor Banner"))
              : (theLogo = GetNamedResource('PICT',
                   "\pB&W Banner"));
   if (ResError() == noErr)
      {
      gThePict = (PicHandle) theLogo;
      HLockHi((Handle) gThePict);
      theFrame = (*gThePict)->picFrame;
      SizeWindow(gBannerWindow, theFrame.right,
                 theFrame.bottom, TRUE);
      DrawPicture(gThePict, (*gThePict)->picFrame);
      HUnlock((Handle) gThePict);
      } /* end if == noErr */
```

Note that you use the Memory Manager routine `HLockHi()`, which moves the data block referenced by `gThePict` high in the application heap before locking it. This helps minimize heap fragmentation. You then unlock the block once

you are done with it so that the application can recover the memory later, if necessary.

For 680x0 Macs, there are lists in *Inside Macintosh* that mention those Toolbox routines that move memory, and thus trigger the memory relocation problems described here. There are other less obvious interactions that Toolbox calls or the Mac OS can do to objects in memory. A good reference work on this subject for the 680x0 Macs is Scott Knaster's *How to Write Macintosh Software.*

Power Macs have a unique set of problems. The most common problem is tripping over unexpected Mixed Mode switches. If the Power Mac seizes up solid, a function probably exists that you failed to provide a UPP for, or you mangled the routine descriptor that's subsequently passed to `CallOSTrapUniversalProc()` or `CallUniversalProc()`.

Debugging Miscellany

When testing fat trap code, you have to ensure that both sections of the trap get called. This is because the Mixed Mode Manager attempts to avoid an instruction set context switch whenever possible. In the case of FlipDepth, the Power Mac's Event Manager is still emulated 680x0 code. Therefore, the Mixed Mode Manager always calls the 680x0 side of the fat trap. To test the PowerPC patches in FlipDepth, I had to compile a PowerPC-only version of the patches to guarantee that the native version of the patch gets called. For "FlipDepth.c," in the declarations area at the start of the file, locate the flag `DO_PPC_CODE_ONLY`. (There's a statement that undefines it located here.) Edit the statement to define `DO_PPC_CODE_ONLY`, and recompile the project with the CodeWarrior IDE. This ensures that only a PowerPC version of the routine descriptor is generated, and not a fat routine descriptor. As an example of this technique, here's the specific code from FlipDepth:

```
/*=======================
   CreateFatDescriptorSys

   Creates a fat routine descriptor
   in the system heap.
=======================*/
```

```
OSErr CreateFatDescriptorSys( void *mac68Code, void *ppcCode, ProcInfoType
➥procInfo, UPP *result )
{
THz   oldZone;
OSErr err = noErr;

   oldZone = GetZone();
/* Save current zone */
   SetZone( SystemZone() );
/* Get you in the system heap */

   #ifndef DO_PPC_CODE_ONLY
   *result = NewFatRoutineDescriptor( mac68Code, ppcCode, procInfo );
   #else
   *result = NewRoutineDescriptor( ppcCode, procInfo, kPowerPCISA ); /* debugging
   ➥only */
   #endif

   SetZone( oldZone );

   return ( *result ? noErr : memFullErr );
} /* end CreateFatDescriptorSys() */
```

It's possible to test the 680x0 portion of a fat binary application on a Power Mac. To do this, open the application with ResEdit and then open the `'cfrg'` resource. There will only be one, with an ID of 0. Select Get Resource Info from the Resource menu, or type ⌘-I. When the Info box appears, change the ID number to something other than zero. Without a `cfrg` resource of ID 0, the operating system is fooled into thinking the application is a 680x0 application, and so it loads and executes the 680x0 CODE resources. To test the PowerPC side of the program, change the `cfrg` resource ID back to 0. Of course, you'll want to test the application on some real 680x0 Macs to eliminate timing and emulator side effects. Testing on a 680x0 Mac can help flush out some improper Toolbox usage bugs as well.

Occasionally you'll want a debugger to break into the execution of an application at certain points. To do this, there are specialized statements that you can add to the program code to cause an exception and invoke a high-level or low-level debugger.

These statements are as follows:

```
Debugger();
/* Break into low-level debugger */
DebugStr("\perror msg");
/* Break into low-level debugger, */
/* display error message */

SysBreak();
/* Break into high-level debugger */
SysBreakStr("\perror msg");
/* Break into high-level debugger with message */
```

The `Debugger()` statement invokes a low-level debugger on both Power Macs and 680x0 Macs. `DebugStr()` accomplishes the same end, but also displays a Pascal-style message string when the debugger kicks in. This message can inform you which `DebugStr()` statement out of several executed. `SysBreak()` and `SysBreakStr()` function as breakpoints that transfer execution from the test program to a high-level debugger, such as MW Debug.

Be aware that the behavior of these statements varies, depending upon your development tools and their settings. For example, MW Debug lets you choose whether it or a low-level debugger takes control when `Debugger()` or `DebugStr()` statements execute. In MW Debug's preferences, you can have it halt whenever a `SysBreak()` or `SysBreakStr()` statement is encountered. You can also configure MW Debug so that it uses the current processor ISA to determine whether it intercepts `Debugger()` statements or forwards them to a low-level debugger. For example, you might have MW Debug take control when a `Debugger()` or `DebugStr()` statement fires in 680x0 code, but it passes these same statements to a low-level debugger when they execute in PowerPC code. If MW Debug isn't running, the low-level debugger always takes control.

Hazard

Even with MW Debug running, you must have a low-level debugger such as MacsBug installed when using these statements. Otherwise, you might crash the system.

Get to know AppleScript. It can help you set up test events for debugging high-level event handlers. It's also useful for writing scripts to automate parts of the development cycle.

Finally, there are a couple of shareware/freeware utilities that can expedite the debugging process. MacErrors, by Marty Wachter and Phil Kearney, is a small application that translates those cryptic error codes into a readable explanation. If a File Manager routine reports a –43, you type this value into MacErrors and press Return. MacErrors explains that the error number means "file not found; folder not found." This message should pinpoint the trouble to that part of your file I/O code that handles a FSSpec or related data structure.

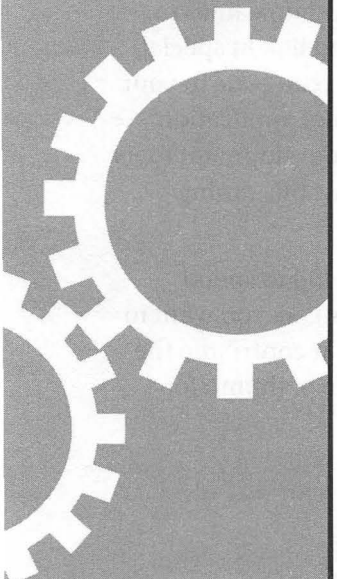
Enough Debugging

In this chapter, you've received an overview of the types of debuggers. You have looked at MW Debug and two low-level PowerPC debuggers for testing your native applications, and you've read about the various types of bugs. Be careful while you write code; you will reduce many of the debugging problems. Proper use of the available debugging tools will minimize the job of finding and eliminating those pesky bugs that do sneak in.

In the next chapter, you learn how to obtain the maximum performance out of your native programs.

Performance and Processors

Most of this book discusses the Power Mac's architecture and how to write native C programs for it. This final chapter discusses how to make your programs better: that is, faster. Some of this discussion will be common-sense code optimizations. However, some aspects of the Mac OS run-time environment and certain processor-specific characteristics exist that can impair a program's performance if you're not careful.



Planning and Profiling

Like writing program code, optimizing code requires some careful thought up front. To begin, you should carefully consider what you're trying to do here. Are you simply trying to make the program more snappy and responsive to the user? Are you trying to boost disk I/O so that the user doesn't become frustrated watching a progress bar slowly crawl across the screen during a file save? Or are you looking for the ultimate in screaming performance, to make a game more playable? Each of these examples points to different solutions.

A more responsive user interface might only require tweaking the code that draws and updates the windows, not a major code overhaul. File operations take some careful consideration because you need to balance the opposing requirements of fast file I/O performance, versus giving the operating system time to handle background tasks and responding to a user's request to abort the operation. Game design requires all the tricks in the book: loop unrolling, heavy-duty buffering, custom-built bit functions, and minimal use of the operating system. In most cases, these solutions might not require fancy code optimization at all, just a careful redesign of certain parts of the program. After you've thought about these issues, you need to do three other things before you consider changing a line of code: profile, profile, profile.

Profiling for a Purpose

Code profiling consists of the addition of special code to a program that measures where it spends most of its time. For example, in the CodeWarrior IDE, you must adjust some preference settings, add several source lines to your program to initialize the profiler and gather results, and link in special-purpose libraries that contain the profiling functions. This extra code in your program gathers timing statistics that the Metrowerks Profiler application uses to display the program's overhead by function. Other development tools are available and have different ways to measure and display this timing information.

Why profile the program's operation? It doesn't do much good to spend hours or days fine-tuning a function that's rarely called. Instead, you want to concentrate your efforts on those heavily used functions that contribute the most overhead to the program's operation. As a general rule of thumb for

code optimization, a variant of the 90-10 rule applies. You only want to optimize the 10 percent of the code where the program spends 90 percent of its time. Any fine-tuning of these key functions delivers a noticeable improvement in the program's operation, making your time well spent. Also, code profiling sometimes uncovers certain logic bugs (perhaps a function being inadvertently called too often) that you'll want to fix anyway. It's quite possible that some of these bugs (especially an often-called function) are the culprit in your program's lackluster performance, so fixing them will help you toward reaching your goal.

It goes without saying that your program should be fully tested and bug-free before attempting performance enhancements. Code optimization efforts unfortunately can introduce their own bugs. By using reliable code as a starting point, when a bug bites you have a good idea where it was introduced.

Performance Issues

Now that you've identified the bottlenecks in your program, you can focus on these key functions. Don't touch that editor yet because you still need to determine why these functions consume most of the time. Is it because this function uses another function that contributes the overhead? Is because the function is compute-intensive? Or is it because the function is called often?

The answer to the first question is obvious: examine the offending function instead. This is why you always profile the program before changing a line of code. The answer to the second question is that you study the compute-intensive code carefully. If the function is making heavy use of the Mac Toolbox, determine whether you can call these routines less often (see next section). Or try to substitute your own routine that accomplishes the same thing. Most game designers, for example, use QuickDraw to set up a port and then use their own drawing functions, rather than QuickDraw, to blast bits into this port. If the compute-intensive function is mostly custom program code, you should definitely optimize it.

The answer to the third question gets tricky. Why exactly is this function called often? If the function is performing file I/O, you might rewrite the code so that the File Manager calls use larger amounts of data, thus reducing the number of calls to the function. Another "called-too-often" problem occurs

with screen updates. While you want controls and status displays to be crisp and responsive, you don't need to update them more than several times a second to achieve this effect. If you're doing this dozens of time per second, you can impair your program's operation by doing needless screen updates. The matter gets worse if these updates involve a Mixed Mode switch. (See Chapter 4.) Finally, if you must update a status display, draw only those items that have changed. As you can see from the varied answers to these three questions, you need to have a good idea of what's causing the bottleneck so that you apply the proper fixes.

Code Tuning

With a clear picture of what functions are dragging down performance, and why they're causing it, now it's time to do the code optimizations. If you're aching to learn how to program in PowerPC assembly, settle down, because that's the method of last resort. You first want to examine the code in the offending function. It's possible a more efficient algorithm, written in C or another high-level language, might do the job. A high-level language that uses an efficient algorithm will always outclass an assembly-language function that implements a poor algorithm. Furthermore, a better-written algorithm in C might boost performance adequately so that you don't have to resort to assembly language programming. This is not to say that assembly-language isn't a solution in some cases. But because assembly language is hard to write and harder to debug, it's better to take your initial stab at code optimization by rewriting better algorithms in C. Finally, from a cross-platform point of view, steer clear of assembly language because your program's portability goes out the window.

What's likely to happen is that you're hankering to rewrite these offending functions anyway. All too often after the program is done, you realize a better way to do things within a particular function. If this function happens to be a proven bottleneck, that's justification to go back and rewrite the code.

Another thing to do is to rethink what the function does. If it's a general-purpose function trying to be everything for every situation, you might want to rewrite it so that the function does a few condition checks and then calls one of several smaller special-case functions. This lets you write tight code in these sub-functions that efficiently deals with each particular condition. At best, you can wind up with several fast functions that together can deal with

every possible circumstance. At worst, you might have a mix of slow and fast functions. If the fast functions handle the majority of the cases that the program deals with, then you've obtained a performance win.

As an example of this rethinking process, the programmers for Marathon (a really cool dungeon-style maze game with 3D effects) originally had a general-purpose texture-mapping engine draw the maze's walls and ceiling. Because the algorithms used to generate textures on the walls were different from those used to handle the floors and ceilings, the game designers decided to special-case these operations. They split the engine's general-purpose texture-mapping algorithms into two separate operations: one for the walls, and another for the ceiling/floors. Next, they fine-tuned each algorithm for its specific purpose. The end result was a faster engine. In addition, this implementation gave the user the option to turn off the floor/ceiling texture-mapping. These objects might look duller with the texture-mapping switched off, but it improved the game's performance. This meant that a larger number of people, some owning slower machines, could still enjoy the game.

A Word about Caches

This book has touched on the fact that much program improvement can be accomplished by use of better algorithms or special-purposing the functions. After you've done that, other areas exist where improved performance is possible. This category of optimizations requires that you understand how a PowerPC system operates.

While the processor in a Power Mac might be blazing along at 150 MHz, the memory subsystem typically is loafing along at a fraction of that speed. A 132 MHz Power Mac 9500's system bus, for example, runs at 44 MHz. In short, any read request to system memory is going to have to wait several cycles or more for the slower DRAM to deliver the goods. Why is this done? There are two reasons. The first reason is that fast buses are expensive and difficult to build. The second reason is that fast memory is expensive. By using slower memory, the system is affordable.

These memory delays are offset by the PowerPC's on-chip caches. *Caches* are special buffers that hold small amounts of the most recently used data. Because these buffers sit on the processor, instruction fetches from the cache

occur at processor speeds. Ditto for the data: if the data set occupies the data cache, these accesses fly. If, however, the processor has to go outside the cache for program code or data, its pipelines can stall waiting for this information. This sort of delay can be reduced by a secondary level of cache memory, called the level 2 (L2) cache. An L2 cache consists of fast RAM (typically with an access time of around 14 nanoseconds, versus 50 to 70 nanoseconds for the system DRAM), which lowers the penalty of an off-processor memory access. Because this fast memory is expensive, L2 caches range in size from 256K to 2MB in size, and are an option on certain systems.

Cache Operation

Both the L2 cache and on-chip caches operate on the assumption that the processor's next read request will be in the neighborhood of the last one. Because a processor typically executes instructions in sequential order through memory, or in loops, this assumption makes sense. The cache logic takes advantage of this situation. When the processor demands the next instruction, the cache's read logic fetches additional instructions from memory addresses adjacent to the target instruction's address. These extra fetches are done to fill buffers within the on-chip cache. When the processor needs the next instruction at the next sequential address, it happens to already be in the cache. This advantageous situation is termed a *cache hit.* Under ideal situations, such as a tight code loop, the processor can execute its repeating cycle of instructions without further accesses to system RAM.

This scheme breaks down when a branch instruction (usually part of a control statement or a function call) requires the processor to jump to another location in memory that's nowhere near the PC's current address. The situation where the cache doesn't possess the target code or data is called a *cache miss.* In this case, a portion of the cache must be refilled with code from the new address. This activity takes time and briefly disrupts the flow in the processor pipelines. It's important to note that the effect of branch instructions can be minimized by the cache. If this particular branch instruction was executed recently—perhaps inside a loop—it's possible that the target code of the branch still occupies the cache.

A RISC processor's operation is analogous to that of a jet engine, in that both machines run smoothly as long as they're fed at a prodigious rate. A momentary break in the data or air flow causes the machine to stall. You can see why

processor designers go through a lot of trouble designing circuitry that predicts the outcome of branch instructions. A mispredicted branch can hamper performance because of the time delay between when the pipelines empty, until instructions at the cache miss address begin entering the pipelines. With today's processors running far faster than the system memory, such disruptions take longer to correct.

Cache Details

Caches wouldn't be very efficient if their entire contents were dumped on a cache miss. Thus caches typically are subdivided into *lines* consisting of a certain number of bytes. The cache line may be further subdivided into separate addressable elements called *sectors*, or *blocks*. An algorithm determines which cache lines are to be replaced with the new data. The PowerPC processor family currently uses a least recently used (LRU) algorithm for this purpose. Future PowerPC implementations might use a different algorithm. Data is read into these cache lines on demand, overwriting the previous contents. (Note: If this line contains a data variable whose value was changed by the program, the processor has to write this new data out to system memory before it is replaced.) This arrangement where the cache is made of groups of lines enables it to hold several pieces of frequently used code or data.

The size of a cache line varies among the PowerPC family, and in some instances the terms line and block describe the same 32-byte element. For the PowerPC 601 processor, a cache line is 64 bytes in size, and consists of two blocks. Each line loads 64 bytes from system memory, starting along a 64-byte boundary (that is, bits 26 through 31 of the effective address are zero).

While the processor deals with the cache at the resolution of blocks, 601 cache operations typically work on a per-line basis. That is, if one block in a cache line gets filled from system memory, the processor attempts to load the other block as well. However, there's no guarantee that the second block will be read in properly. The 601 has a 32K unified cache that can hold both code and data.

The 603 and 604 implement a Harvard architecture that uses separate caches for code and data. In these processors, the caches consist of 32-byte blocks.

The size ranges from 8K (603) to 32K (604e) in size. Blocks are loaded from a 32-byte boundary in memory (that is, bits 27 through 31 of an effective address are zero).

Another thing to keep in mind is that even under the best conditions, a data set or a function might not fit into a cache. The size of the caches varies among the PowerPC processor family, with the worst-case situation being the PowerPC 603, which has 8K caches. This size typically is adequate for most code. Programs usually work with the large amounts of data, so you might think a small cache wouldn't offer a performance win. However, stack variables tend to hit very well in a data cache. Structure and vector operations also do well, even if they aren't reused, because cache line fills bring in the subsequent elements before they get referenced. Finally, by holding critical control variables in a cache, it's possible to improve the performance of certain operations.

Important

So far, this discussion has covered reading information into the cache. There are other cache issues related to writing data to a cache, especially to keep the contents of system memory up-to-date, or *coherent*, with the cache's contents. Usually the system memory is updated on-the-fly automatically, and you need not worry about how this is done. Certain situations exist, however, where this problem requires that the program have explicit control of the cache. This is the sort of thing that only folks writing operating systems or device drivers need worry about. Such details are beyond the scope of this book.

Before you read about optimization techniques, it's worth mentioning that you should optimize one function, and then run the program through the profiler again. The Power Mac environment is a complex one, and it's possible that what seems like a reasonable code optimization might actually make things worse. This is particularly true if your function makes several calls to emulated Toolbox code, or somehow causes numerous cache misses. As you gain a better understanding of the system's behavior, you'll write better code.

General Caching Principles

Because accessing system memory can be expensive from a performance perspective, a rule emerges: *Make your requests work with the cache.* The geometry of your program's logic flow can affect its performance by reducing cache misses. How you accomplish this depends on whether you're working with code or data.

For code, when you write loops, keep them small if possible. This way they stand a good chance of fitting in the code cache. When using `switch` statements to handle decision paths (such as in the `main` event loop), keep them compact. Instead of writing a big switch statement crammed with lots of inline code, write a compact one that calls functions. This reduces the possibility of cache misses occurring when the flow of execution tours the switch statement. Of course, there will be a cache miss when the `switch` calls a function, but the point is to keep cache misses from occurring on the decision path up to the function call.

As a counterpoint to this rule, don't be too liberal about piling everything into functions. If a loop repeatedly calls a function that does a simple operation, it's worth moving that function's code within the loop. By adding the code inline to the loop body, the overhead of a function call goes away (see Chapter 4 about what a function call involves), as does the possibility of its branch instruction creating a cache miss. Note that functions don't have to be inside a loop to be a candidate for inlining, just small and simple.

If you're writing C++ code, you can use the `inline` keyword to assist you with function inlining. Start by declaring the candidate function `inline`, like so:

```
inline long High_Use_Func()
{
// small section of program code
;
} // end High_Use_Func()
```

With the `inline` keyword in place, go to the CodeWarrior IDE'S C/C++ Language preferences panel and confirm that the Don't Inline item is unchecked. If the function meets certain requirements, the Metrowerks C++ compiler places this code inline within the calling function's code. Consult the *Metrowerks C/C++/ASM Lang Ref* for the criteria the compiler uses to determine whether it can inline a function. This arrangement gives you code

inlining without having to do a massive edit to your source code. However, be aware that code inlining can create serious code expansion, particularly if a key function is called dozens of times throughout the program. Be sure to run the program through the profiler again to evaluate the results.

If you're using a jump table to control program flow, try to keep it small to minimize cache misses. An example of the problems that can occur with a large jump table is Apple's 680x0 emulator. This jump table functions as a 680x0 instruction decoder, but it flooded the PowerPC 603's small code cache. For the Power Mac 5200, Apple's workaround was to integrate 256K of level 2 cache RAM onto the Mac ROM SIMM. The PowerBook 5300 and Duo 2300 use the PowerPC 603e, which has a larger (16K) code cache. To be fair, the 68040 has about 200 instructions, with numerous variations brought about by eighteen different addressing modes. A large jump table was probably the best way to implement a high-speed decoder.

Where possible, try to keep seldom-used code out of the direct flow of the program. Good subjects for this rule are error-handling code blocks. If an operation fails (say, you can't allocate some memory), rather than have the code that displays the error dialog box following the statement that does the condition check, have it call an error-reporting function instead. Otherwise, the processor has to fetch lots of rarely used code as part of the program's normal operation, and this can cause cache misses to occur. This is the sort of stuff you'd do anyway to make a well-organized, readable program, but it doesn't hurt to emphasize that it can affect performance as well.

A final tip is to group your PowerPC functions according to their purpose. You can do this using the `#pragma segment` directive available in the Metrowerks C/C++ compiler. Here's a code fragment that illustrates how you arrange a program's functions:

```
#pragma segment setUpModule
OSErr Init_Memory()
{...}
OSErr Init_Draw(grafPort)
{...}
OSerr Init_Files(volNum)
{...}

#pragma segment drawModule
long Brush_Tool(grafPort)
{...}
```

```
#pragma segment printModule
long Print_Window(grafport)
{...}

#pragma segment drawModule
long Pen_Tool(grafPort)
{...}
long Eraser_Tool(grafPort)
{...}

#pragma segment fileModule
OSErr Save_Window(volNum)
{...}
OSErr Import_TIFF(volNum)
{...}
Read_Disk(volNum)
{...}
Write_Disk(volNum)
{...}

#pragma segment printModule
long Get_Printer()
{...}
```

The functions `Init_Memory()`, `Init_Draw()`, and `Init_Files()` are combined in a group called `setUpModule`. `Brush_Tool()`, `Pen_Tool()`, and `Eraser_Tool()` hang out in the `drawModule` group, while `Get_Printer()` and `Print_Window()` are in `printModule`. Finally, `fileModule` contains `Save_Window()`, `Import_TIFF()`, `Read_Disk()`, and `Write_Disk()`. When the linker generates the code fragment, the code for these functions is organized by group name, rather than by where the functions appear in the source file. Now when the Mac OS loads the code fragment, these similar functions are placed in proximity to one another in memory. For example, when this hypothetical program wants to write a disk file, all of the I/O functions it uses (`Save_Window()` and `Write_Disk()`) are clustered together in memory. Although all of the functions probably can't fit in the processor caches, it's possible that they will land in the L2 cache. This shaves a few clocks off a cache miss when the processor obtains the target function's code in the faster L2 cache RAM instead of system memory.

Important

For the PowerPC processor, the `#pragma segment` doesn't place native object code in anything that corresponds to 680x0 code segment. It only determines how the object code gets arranged inside a code fragment.

Also, for the CodeWarrior IDE to honor this `#pragma segment` organization, you have to check the item Order Code Section by `#pragma segment` in the PPC PEF preferences panel.

Data handling has a different set of issues. Basically, you want to avoid making read requests that are scattered all through memory. Instead, do them in bulk, and preferably from a contiguous set of addresses so that the processor can fetch all of them at once through filling cache lines. This almost guarantees that subsequent accesses to these variables hit the cache. For this reason, when you write data structures, keep the most often-used variables for a particular operation clustered together so that they occupy a cache line.

Watch your use of global variables. Because they're located elsewhere, typically in the code fragment's TOC, they can create a cache miss. If the global variable happens to be a state flag that isn't going to change rapidly, make a copy of it into a variable local to the function and use the copy instead. This way you're only penalized once for the cache miss accessing the global variable that one time.

If you're using a look-up table of data values to speed a computation, reconsider using the PowerPC floating-point instructions. They're pretty fast, and they might provide the desired accuracy. This way you get a more compact function code-wise, and avoid the use of a large look-up table that can create all sorts of cache misses.

Finally, try using large block sizes when moving data around. The sequence of load/store operations used to copy the contents of buffer A into buffer B can cause a pipeline stall. By moving more data with each load/store operation, you can reduce the effects of the stall. Also, if you cast the pointers so that they reference a `float` data type, you can get the FPU to assist you by having it move 8 bytes of data at a time.

The 604 processor has two special registers whose purpose is to count basic events within the processor. Apple has released a tool called 4PM that uses these registers to supply various processor statistics. These numbers can be used to help you profile a program's activity. Three of these events, mispredicted branch instructions, instruction cache misses, and data cache misses, are of interest to you. 4PM thus enables you to monitor cache activity so that you can fine-tune your program to minimize cache misses. Your program's cache usage will be similar on other PowerPC processors, so any adjustments made on a 604-based system should be applicable to all Power Macintoshes.

Simple Optimizations

So far, you have read about how the arrangement of your program code can affect performance. Now let's look at how your development tools can do code optimization for you. First and foremost, get your compiler on the job. It can do a large number of code optimizations for you. Best of all, many of these optimizations are free in that you just change some compiler settings and recompile. If your performance goals are modest, a recompile alone might do the job.

Another approach to this problem is to use one compiler to write the code, and another compiler to generate the final executable. Some programmers use CodeWarrior to write a program because it's fast and automatically handles a lot of the bookkeeping chores. Then they use Apple's MrC or Motorola's PPC SDK to do the final compile, because these compilers generate very efficient code.

Instruction Scheduling

Each PowerPC processor has a certain number of execution units that operate in parallel to boost performance. The PowerPC has three execution units; the 603, 603e, and 166 MHz 603e have five; and the 604 and 604e have six. (Note that one of the execution units in the 603 family handles certain system- and power-management functions, and so you may consider that this family actually has four execution units.)

Suppose, however, that all the code the processor fetches and executes is floating-point instructions. In this case, the floating-point execution unit

backs up, while the integer and branch execution units sit idle. Ideally, you want to order (or schedule) a mix of different instructions throughout the program's operation so that all of the processor's execution units remain busy, rather than having one execution unit become a bottleneck. Practically, such an ideally scheduled stream of code doesn't occur often. Again, the compiler can help you out here. As it generates processor instructions from your source code, the compiler can also shuffle them about to balance the load to the various execution units.

To initiate instruction scheduling in the CodeWarrior IDE, go to the PPC Processor preferences panel. Under the Optimizations section, use the Instruction Ordering popup menu to select 601, 603, or 604. Now when you make your program, the compiler attempts to schedule the code sequence for the chosen processor. This way, say, the 603 processor encounters a blend of different instructions that it can dispatch to every execution unit. The default setting for this item is None, because rearranging the instruction sequence also obscures the program's operation, which makes it hard to debug. Take the hint from this and ensure that your program is tested and debugged before applying any optimizations to it.

Although compilers are very good at rearranging the instruction stream, there are things that you can do to help. Consider converting some of your integer-math algorithms so that they use some floating-point math. This gives the compiler an opportunity to schedule a mixture of integer and floating-point instructions. Be careful about doing this if you plan to convert data to the floating-point format and back. Although the PowerPC has two instructions for converting float-point values into integers, there are no corresponding processor instructions for converting integers into floating-point values. These conversions must be implemented in software, and such overhead can negate the benefits of the better instruction scheduling. Your best bet is to rewrite the algorithm slightly so that certain variables remain as floating-point values throughout the program. This is one of those areas where you have to use a profiler to see if your modification improves the situation.

Another trick to improve instruction scheduling is loop unrolling. Say you have a code loop that looks like this:

```
for (i = 0; i < 300; i++)
    {
    red[i]= 0xFFFF;
```

```
    green[i] = 0xFFFF;
    blue[i] = 0xFFF;
    } // end for
```

You can "unroll" the code so that the loop looks like this:

```
for (i = 0; i < 150; )
    {
    red[i]= 0xFFFF;
    green[i] = 0xFFFF;
    blue[i] = 0xFFF;
    i++;
    red[i]= 0xFFFF;
    green[i] = 0xFFFF;
    blue[i] = 0xFFF;
    } // end for
```

This is a very simplistic example, used to illustrate the technique. Such loop codes normally perform some scary computations. By unrolling the loop, the redundant code presents a larger number of mixed instructions to the compiler. It can then organize these instructions into a balanced sequence that keeps the processor's execution units busy.

An added benefit to loop unrolling is that the new code reduces the number of iterations the loop performs. This in turn reduces the effect of the branch instruction that takes the flow of execution back to the start of the loop. The farther the branch instruction that ends the loop is moved away from the compare instruction that starts the loop, the less impact the branch instruction has. Be careful, however, about unrolling loops too much. The code expansion this technique produces might create cache misses. A general rule of thumb is to not unroll the loop more than four times.

These are a few suggestions for improving the performance of your programs without resorting to assembly language. Your best course of action is to try some compiler settings, add code improvements in small increments, and measure the outcome every time with a profiler.

Processor Specific Issues

One of the purposes of the PowerPC 601 processor was to serve as a bridge that let developers migrate programs from a variety of processors, such as POWER (IBM workstations running AIX), 680x0 (Macintosh running Mac

OS), and x86 (systems running Windows NT). Because of the myriad ways these processors organize data in memory, the 601 was designed to be flexible in how it accesses memory. One such capability was that the PowerPC 601 (and all members of the PowerPC family) handles bi-Endian addressing modes. That is, it can fetch data regardless of its memory organization.

Background Info

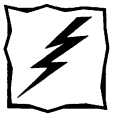
Endian addressing modes determine how data bytes get placed in memory. This affects the organization of data quantities larger than a byte. This book uses a 16-bit word or short as an example, but the situation applies equally to larger data types such as longs and floats. Processors such as the Motorola 680x0 and IBM's POWER store the most significant byte (MSB) in the lowest byte address, whereas the least significant byte (LSB) of a word quantity occupies the higher byte address. This memory organization is known as Big-Endian addressing. Intel's x86 processors place the word's LSB at the lower address, while the MSB gets stored in the higher address. This memory arrangement is known as Little-Endian. There is no performance benefit to either memory addressing scheme. It does, however, make porting say, a Little-Endian Windows NT program to a processor using a Big-Endian operating system difficult because the way the processor arranges bytes in memory is "backwards" from how the NT program expects it. By supporting both addressing modes, the PowerPC processor allows x86-based programs to be ported rapidly to it.

The PowerPC processor actually offers the capability of running both Little-Endian and Big-Endian programs simultaneously through the use of separate address mode bits for both programs and interrupt handlers. The overhead involved in switching between the two addressing modes, however, is so large that such a feat isn't practical for the time being. However, this capability allows a PowerPC Platform system to run different operating systems at different times. You might run a Little-Endian OS first, then you can restart the computer into the Big-Endian mode and run the Mac OS.

Another one of the 601's capabilities is that it is tolerant on how data is positioned in memory. As anyone who has programmed the 680x0 processor knows, it's particular about having data types larger than a byte

word-aligned. (That is, the memory location occupied by a variable that is a `short` or `long` in size must start on an even address.) PowerPC processors favor memory alignment that corresponds to the data's size. They access bytes at any address, `short`s (16 bits) at every even address, and `long`s (32 bits) at every address divisible by four. While the PowerPC actually can access data elements at any address, those that follow the preferred alignment scheme require fewer bus cycles to fetch than unaligned ones. Because emulator programs have to fetch a non-PowerPC program's instructions from anywhere in system RAM, the PowerPC 601's bus fetch circuitry was designed to be flexible and fast in handling misaligned data accesses.

The PowerPC 603 and 604 demand that your program structures respect data alignment. They can handle misaligned data, but at the expense of using more clock cycles. Worse, when these processors operate in the Little-Endian addressing mode and software accesses misaligned data (such as when a 4-byte quantity straddles a 2-byte boundary), an exception occurs and a millicode exception handler fields the access.



Important

Microcode is an on-chip program that decodes various processor instructions and operates the appropriate sections of the processor's logic to execute the requested action. Microcode can be considered a computer program embedded inside the processor. In complex processors, separate programs, called *nanocode*, operate their own sections of the processor, such as a floating-point unit and the integer unit.

Millicode, as its name implies, operates at a higher level, outside the processor. It implements highly efficient routines for frequently used functions. As you saw in Chapter 4, a PowerPC function uses a set of calling conventions, such as placing arguments in processor registers and adjusting either the branch or link register to point to the target function's address. To reduce overhead, millicode doesn't follow these conventions. Instead, it uses the branch absolute and link instruction (`bla`). This reduces the function call overhead to that of a single `bla` instruction to enter the handler, followed by a register-based branch (using the `link` register) to return when the function exits. Such unconditional branches typically take zero cycles to execute because the processor's branch unit can resolve them well in advance.

continues

continued

The 603 and 604 processors use millicode to handle certain misaligned data accesses, and to resolve the byte order in Little-Endian accesses. (Big-Endian byte-ordering is handled by the processor hardware.) As efficient as millicode is, it can add substantial overhead if the handler is called frequently.

With the 166 MHz PowerPC 603e and the PowerPC 604e, the hardware now keeps track of the byte ordering for both Big- and Little-Endian addressing. Because this eliminates the overhead of a millicode handler, load/store operations now take the same number of cycles to execute regardless of the Endian addressing mode on these processors.

The upshot is that misaligned data can impair your program's performance because it slows the completion of read requests for 603 and 604. In certain situations, the performance degradation can be substantial. The 603e and 604e have improved load/store logic that reduces this misaligned access penalty. However, just as you should tailor your code with respect to the smallest cache size, you should be proactive and write your code keeping data alignment in mind. This lets your program work with any processor to achieve optimal throughput. Fortunately, by using the proper compiler settings, you can align your program's data so that many of these problems are eliminated.

You can head off trouble caused by misaligned data a number of ways. After each description I'll offer some possible fixes. As usual, after you've made a change to the program, measure the results with a profiler to determine whether it has the desired effect.

The roots of one source of data alignment trouble is historic. Developers who ported 680x0-based Mac programs to the Power Mac kept their data structures aligned on 2-byte boundaries because that was the preferred data alignment for that processor. On the 601, this minimalist coding approach incurred little or no loss in performance. On the 603 and 604, however, the misaligned data can exact a heavy performance hit, making the program up to 40 times slower under the right circumstances. The solution is to modify the program's data structures so that they present an optimal alignment for

the PowerPC processor. Remember that `short` data types should be aligned on 2-byte address boundaries, and `long`s should be aligned on 4-byte address boundaries. You can have the Metrowerks PPC compiler handle these details automatically by going to the PPC Processor preferences panel and choosing PowerPC from the popup menu for the `Struct` alignment item. Alternatively, you can bracket key data structures with `#pragma options align=power` and `#pragma options align=reset` statements so that the compiler applies PowerPC data alignment to them.

Another gotcha is mixing up different data types in your structures. Suppose you write the following data structure:

```
struct rgbBlock
{
short        seed;
Boolean     redIsDirty;
long         red;
Boolean    greenIsDirty;
long         green;
Boolean    blueIsDirty;
long         blue;
short        colorSpace;
Boolean    hasAGWorld;
}
```

A PowerPC compiler will try to organize each element in this structure along its desired memory boundary. That is, it will place `red`, `green`, and `blue` on a 4-byte boundary, `seed` and `colorSpace` on 2-byte boundaries, and so on. It does this by inserting padding bytes after certain elements to reposition the others in memory. For example, it will add several padding bytes after `redIsDirty` to position `red` on a 4-byte boundary. The size of this structure is 28 bytes. If, however, you reorganize `rgbBlock` like in the code fragment that follows, every element falls on its proper alignment boundary, and the compiler doesn't have to supply padding bytes:

```
struct rgbBlock
{
long         red;
long         green;
long         blue;
short        seed;
short        colorSpace;
```

```
Boolean     redIsDirty;
Boolean    greenIsDirty;
Boolean    blueIsDirty;
Boolean    hasAGWorld;
}
```

This might seem like a minor issue, but it can have important consequences in complex structures. First, the smart arrangement of data elements conserves memory. The revised version of `rgbBlock` now requires only 20 bytes of memory. Second, from a performance standpoint, by eliminating extra padding bytes, the structure is more likely to fit in a cache line. It didn't happen in this example, but suppose you have a structure that becomes 33 bytes in size if the compiler adds padding bytes? You'll suffer a huge performance hit when a heavily used data structure causes frequent cache misses. Bear in mind that you're working with a structure used for a specific purpose here. Don't attempt to organize all your program variables by type, because in moving the variables about, you might unintentionally create cache misses. Organize data elements by purpose and frequency of use first, then arrange them by data type.

When copying large blocks of data, try to ensure that both the source and destination buffers are properly aligned in memory. You might consider using `BlockMove()` to handle the job, although a custom special-purpose move function might execute faster. Again, try a solution and profile the result.

Finally, you'll have to decouple how you handle data in memory from how you handle data on a disk drive. Some programmers took C structures in memory and wrote them to disk as a stream of bytes. To reconstitute these structures at a later date, all they did was read the bytes from disk back into memory. However, such structures can contain misaligned data elements. The fix is to write wrapper code that optimally organizes the data for the target medium. Data transfers into memory would be properly aligned, and data written to the disk would be in a format that conserves disk space.

As was mentioned in Chapter 5, be careful if you expect this program to run on a 680x0-based Mac, or share data with them. Data structures optimally aligned for a PowerPC-based system might cause an exception when used on a 680x0 processor. There are two solutions. First, live with using 2-byte aligned data structures, and take the performance hit. Or, use the wrapper code just described to properly align the data for the host processor.

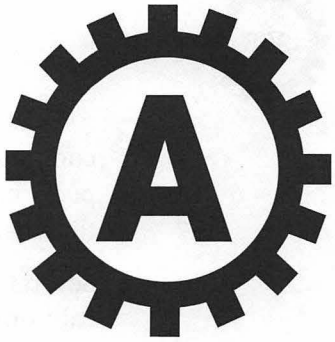
Important

As a last resort, you can use the Gestalt Manager (which you met in Chapter 5) to obtain hardware-specific information on the Power Mac running your program. You shouldn't use this information to control your program's operation; it creates problems because you can't anticipate what results future (and faster systems) might return. The `'bclk'` and `'pclk'` selectors are available only on systems running System 7.5.2 or later. If you must, here's the processor-specific data:

Gestalt Selector	*Purpose*	*Return Value*
`cput`	CPU type	0x101 = PowerPC 601 0x103 = PowerPC 603 0x104 = PowerPC 604 0x106 = PowerPC 603e
`bclk`	Bus clock speed	Bus clock speed in Hz.
`pclk`	Processor clock	Processor clock speed in Hz speed.

Summary

Your journey has at last come to an end. I hope your trip with me has been an interesting one. If you learned something of value between these pages, my trip with you has been rewarded.



Appendix A: The PowerPC RISC Processor Family

The first-generation Power Macintoshes are based on the PowerPC 601, a RISC microprocessor jointly developed by Apple, IBM, and Motorola. Later Power Macs use the PowerPC 603 to address a low-cost market, or they use the PowerPC 604 to target high-performance work. The PowerPC processor family is designed to be a low-cost processor architecture that supports a wide range of applications from embedded applications (such as in automobiles) to hand-held Personal Digital Assistants (PDAs) and desktop computers. This processor uses a high-performance processor core; other portions of the design provide versatility by being tailored to the target application. For example, a PowerPC fabricated for a desktop computer might have a large cache, and a PDA version might reduce the cache size and eliminate multiprocessing features to minimize power consumption.

Currently, four members make up the PowerPC family, with variants that address specific needs such as larger caches. The previously mentioned PowerPC 601 targets mid-range desktop computers, such as Apple's Power Macs and IBM's Power Personal desktop systems. The PowerPC 602 is a low-power version of the architecture, tailored specifically for hand-held devices. The PowerPC 603 is a low-power implementation of the PowerPC processor that's optimized for use in notebook and sub-notebook

computers, and low-end desktop systems. The PowerPC 604 is a high-performance processor with larger caches and smart branch prediction logic that makes it suitable for workstations.

Several elements of the PowerPC design enable it to achieve the diametrically opposed goals of high performance and low cost. First, the RISC design facilitates high instruction throughput. By using basic, fixed-length instructions, RISC processors have a simple hardware instruction decoder that can dispatch instructions in one clock cycle. This differs from the Complex Instruction Set Computing (CISC) processor, whose decoder is more complex and requires several clock cycles to read in variable-length instructions and dispatch them.

Important
When the processor dispatches an instruction, the decoder passes the translated instruction to the appropriate sections of the processor for execution. These sections, which are organized around the instruction's purpose (such as integer math, floating-point math, and program branches), are called *execution units*. Note that while it takes only one clock cycle to dispatch an instruction, it might take one or more clock cycles for the instruction to actually execute.

The RISC design also uses pipelining to improve instruction throughput. A *pipeline* is where the instruction's actions are broken into several stages inside the execution unit. To illustrate this, suppose the decoder dispatches the translated instruction to the pipeline in, say, a floating-point execution unit. Each stage in the floating-point unit's pipeline handles a portion of the instruction's execution. For example, the first stage of the floating-point pipeline might obtain the first number from a register; the second stage would obtain the second number from another register; the third stage would perform the calculation; and the fourth stage would write the result back into a register.

Pipelines improve throughput by processing several instructions at once, where each instruction is at a different stage of execution in a different section of the pipeline. As long as the various pipelines are kept filled, instruction processing occurs at a constant rate. Under ideal conditions

when the processor's on-chip cache keeps the pipelines full, one instruction completes execution for every tick of the processor clock.

The PowerPC processor architecture also uses multiple execution units. Furthermore, the instruction set was carefully designed so that most instructions don't overlap, or depend, on other instructions. Otherwise, the flow of instructions in a pipeline might be interrupted or *stalled* because one of the instructions has to wait on a result from an instruction in a different pipeline. This way a floating-point unit can work concurrently on its floating-point instructions as an integer unit works on its set of instructions.

To reduce design costs, the PowerPC architecture was based on IBM's POWER (Performance Optimization With Enhanced RISC) 64-bit architecture. This decision gave the PowerPC designers a ready-made instruction set and RISC processor core for the chip. The PowerPC architecture differs from POWER in its support for multiple processors and single-precision (32-bit) floating-point instructions. (POWER's 64-bit floating-point instructions are also supported.) The PowerPC 601 implements most POWER instructions (certain complex or nonscalable POWER instructions were deleted), and thus a host of IBM software development tools was immediately available to write PowerPC software.

Another cost reduction became possible because the initial PowerPC processor bus is based on the bus of the Motorola's 88110 RISC processor. This bus has high throughput and also supports multiprocessing. This decision provided another ready-made portion of the PowerPC design.

IBM and Motorola have boosted the performance of these processors using different process technologies, higher clock rates, larger pipelines, larger caches, and more execution units. The following sections examine the current members of the PowerPC family.

PowerPC 601

The PowerPC 601 packs 2.8 million transistors onto a die that's 132 mm^2. It's fabricated using a 3.6 volt, 0.65-micron four-metal-layer CMOS process. Early versions of the 601 operate at clock speeds from 50 MHz to 80 MHz. At 66 MHz, the 601 dissipates 9 watts of power, peak. Faster versions of the 601 use a 0.5-micron process that reduces the die to 74 mm^2 and lowers power consumption to 4 watts.

The 601 is a 32-bit implementation of the 64-bit PowerPC architecture. It has a 32-bit address bus that can access 4G of physical memory. A built-in Memory Management Unit (MMU) supports 52-bit virtual addresses. The 601 supports 64-bit data and has a 64-bit data bus. It has a massive 32K on-chip unified cache. The term *unified* means that both data and code occupy the cache. Additional buffers and arbitration logic are required to keep both data and code moving in and out of the cache. Three independent execution units (integer, floating-point, and branch unit) allow up to three different types of instructions to execute at once on the 601.

The 601 can be viewed as a bridge chip for moving from the POWER architecture to the PowerPC architecture. For IBM, POWER workstation applications can be migrated quickly to PowerPC systems. Current Power Personal desktop systems also support other operating systems such as Windows NT and AIX. It is also a bridge for Apple's shift from CISC to RISC computing. It supplies formidable processing power, enough to operate the 68LC040 emulator that makes much of the Power Mac's system software possible.

PowerPC 603

The PowerPC 603 is the 601's low-power sibling. It uses a 3.3 volt, 0.5-micron four-metal-layer static CMOS technology to place 1.6 million transistors on a die 85.1 mm^2. At 3.3 volts and 80 MHz, the 603 dissipates 3 watts, peak.

Like the 601, the 603 is a 32-bit version of the PowerPC architecture, with a 32-bit address bus and 64-bit data bus. The 603 also uses the same pipelined architecture and thus is able to dispatch three instructions at a time.

The 603 differs from the 601 in several ways. First, it uses a Harvard architecture, where data and code are handled separately. It has two independent 8K caches—one for code and one for data—each with its own MMU. The smaller cache size is offset by the reduced complexity of the circuitry required to manage the caches. The arbitration logic needed to manage the 601's unified cache is gone, and the temporary buffers are smaller. The net result is that the 603 musters nearly the same performance as the 601 while using fewer transistors. Also, because the 603 is expected to be used in small, portable systems, the multiprocessor support has been stripped from the design.

Next, the 603 has five, rather than three, execution units. It's important to note that these two extra units provide support functions to manage the energy-saving features and data transfer rather than execute instructions. It still has the same integer, floating-point, and branch units. The first new execution unit is a load/store execution unit that manages data transfers between the data cache and various registers. It executes the load and store instructions, thus freeing the integer unit from the burden of computing effective addresses. The other execution unit is a system register unit that handles the power-saving functions in the 603.

The 603 uses static logic, so the contents of registers and the caches are preserved even when the clock to the processor is stopped to conserve power. The 603 provides three different power-saving modes that implement different levels of energy consumption. These modes are under software control. Dynamic Power Management (DPM) logic switches off idle sub-systems or execution units. The power management logic watches the instruction stream and powers up an idle unit—say, the branch unit—on an incoming branch instruction.

Finally, the 603 has a phased lock loop (PLL) clock multiplier circuit. This enables the 603 to operate reliably even though the system clock might be slowed to reduce a notebook computer's overall power consumption. Also, it acts as a multiplier so that the processor can operate at 66 MHz internally, while the rest of the system runs at 33 MHz.

The 603's low power consumption, combined with its near 601 performance, makes it suitable for notebook designs. Because it is nearly code-compatible with the 601, applications written for Power Macs or Power Personal systems should run on these low-power systems with little or no modifications. The PowerPC 603 lacks certain POWER instructions in order to conserve die space. Therefore, exercise caution with programs that rely heavily on POWER instructions when moving them to this processor and the Power PC 604.

603e

The 603e is, simply put, an enhanced version of the 603. It's a 3.3 V part, and uses a 0.5-micron four-level-metal static CMOS technology. It packs 2.6 million transistors on a die that's 98mm^2. The obvious enhancement is that

the 603e supports a faster clock: it operates at 100 MHz to 120 MHz, while the 603 tops out at 80 MHz. The processor also supports a wider range of clock multipliers, which allows systems designers to hold down power system consumption by using slower clock rates in the system. Like its predecessor, the 603e has DPM logic that manages the activity of the various subsystems. At 100 MHz, the 603e typically dissipates 3 W.

The most prominent change to the overall processor design is the large on-chip cache size: it has two separate, 16K four-way set associative caches while the vanilla 603 has two 8K, two-way set associative caches. The larger caches are possibly in response to Apple's reported difficulties in getting its 680x0 emulator to function on the 603. This emulator uses the 680x0 opcode as an index into a large look-up table that points to the corresponding PowerPC instructions. It was this large table that flooded the 603's caches. Another enhancement is the optimized load/store instructions: they now take only a single cycle to execute. Saving one cycle might not appear significant, but because a processor is either executing instructions or fetching and writing data, this savings adds up to better overall performance.

166 MHz PowerPC 603e

This enhanced 603e design was called the 603++ or 603ev, before the engineers settled on using the clock speed to differentiate this part from its predecessors. It's a 2.5 V part made with a 0.35-micron five-layer-metal static CMOS process. The 166 MHz 603e contains 2.6 million transistors, approximately the same as the 100 MHz 603e, packed onto a die that's 81mm^2. This makes it smaller than the original 603e. Despite operating at the higher clock rate, the 166 MHz 603 consumes only 3 W at 166 MHz, identical to a 603e running at 100 MHz. The design achieves some of its reduced power consumption by operating the processor core at 2.5 V while the bus and I/O interface still operate at 3.3 V. It's pin-compatible with the 100 MHz 603e.

A modification to the 166 MHz 603e's load/store logic provides better performance and support for little-endian addressing modes under Windows NT. Formerly, when the PowerPC operated in little-endian mode and software accessed misaligned data (such as when a 32-bit word straddles a 32-bit word boundary), an exception occurs. A software exception handler then fields the access. Put another way, the processor first had to perform two accesses to read data crossing a word boundary. The lower-address word was

accessed first, regardless of the memory addressing mode. The processor then spent additional cycles in the exception handler that determined the endian order of the data. In the 166 MHz 603e, the hardware keeps track of the data order. With the overhead of a software handler absent, misaligned data accesses complete several cycles faster. As a result, move operations now take the same number of cycles, regardless of the endian addressing mode.

PowerPC 602

The 602 is a 3.3 V part fabricated using a 0.5 micron four-level-metal static CMOS technology. The die measures 7.07 mm^2, and contains one million transistors—making it smaller than the 603. The 602 implements a 32-bit version of the 64-bit PowerPC RISC architecture, where the processor supports 32-bit addresses and 64-bit data. However, where the 603 has separate data and address pins, the 602 time-multiplexes the address and data signals on one set of bus pins. This enables the 602 to be housed in a 144-pin plastic QFP, while the 603 uses a 240-pin ceramic QFP. Although this trade-off requires extra bus cycles for data accesses, IBM and Motorola expect the 602's bus to out-perform any memory subsystem using 70 nanosecond or slower RAM. Its small die size and fewer signal lines make the 602 attractive for low-cost applications, where design issues of price and logic board real estate are critical.

The 602 has two separate on-chip caches for instructions and data. These caches are 4K in size, and each is managed by a separate MMU. The smaller cache size is balanced by the performance of the cache's two-way set associative organization. Like the 603, these caches support a three-state cache coherency protocol (modified, exclusive, and invalid) that's tailored for a stand-alone system design. The 602 has four independent execution units (integer unit, branch processing unit, load/store unit, and floating-point unit). The 602 lacks the system unit found in the 603. Because the 602 is expected to operate in single-user environments such as in PDAs, the OS services that this unit managed were deemed superfluous by the designers.

The 602's floating-point unit (FPU) handles only single-precision (32-bit) IEEE-754 standard arithmetic, while the 601, 603, and 604 handle both single- and double-precision (64-bit) arithmetic. Software emulation routines support double-precision calculations where necessary.

The 602 uses static logic, which preserves the internal state of the caches and execution units when the clock signals to these devices are disabled. The 602 uses the same power-saving modes as those found in the 603: doze, nap, and sleep. The doze mode switches off most of the processor except for the bus snooping logic, which maintains the coherency of the internal caches. The nap mode disables the bus snooping, for further power savings. The sleep mode disables the clock to all internal units, for maximum power conservation. Even operating at the full power, the 602 uses the same dynamic power management (DPM) techniques found in the 603.

PowerPC 604

The PowerPC 604 is the high-performance member of the PowerPC family, featuring larger caches, smart branch prediction logic, and more exeuction units. It's a 3.3 V part, made from a 0.5-micron four-level-metal CMOS process. It has 3.6 million transistors packed on a die 196 mm^2. At 100 MHz, the 604 dissipates less than 10 W. It supports several different bus multipliers so that the system bus can operate at lower clock speeds while the processor itself runs at the maximum clock speed.

The 604 is a 32-bit implemenation of the PowerPC architecture, with a 32-bit address bus and 64-bit data bus. It has two separate 16K four-way set-associative data and code caches. To boost performance, the 604 has six execution units: three integer, an IEEE 754-compatible floating-point execution unit, a load/store unit, and a branch unit. Two of the integer units handle single-cycle register-to-register instructions. The third unit handles more the complex integer multiply and divide operations. The 604's decoder can dispatch up to four instructions per clock cycle. Each execution unit is fronted with a two-stage reservation station. The purpose of the reservation station is to prevent a stalled execution unit from blocking the dispatch logic from issuing instructions to other execution units. When the first stage of the execution unit clears, the reservation station issues the queued instructions.

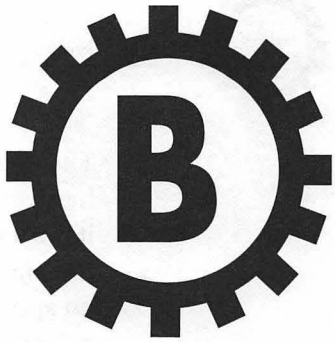
The 604 uses dynamic branch prediction to minimize the delays that occur when a branch instruction alters the course of the program's flow. This is unlike the 601 and 603 processors, which use a hint bit in the branch instruction's coding to determine the direction of the branch. The 604

ignores the hint bit, and instead creates a branch history table to predict the course of the current branch instruction. This history table supplies a branch target address cache with both the address of a branch instruction and the target address of the branch. The fetch logic accesses this cache as it operates. If a fetch address matches an address in the cache, the branch target address in the cache is used rather than the current fetch address. Subsequent instructions are executed speculatively (that is, they don't update the processor's internal state) until the outcome of branch instruction is actually resolved. If the 604's prediction is accurate, the overhead of a program branch is only a few cycles. If the prediction is wrong, the processor can flush all of the six execution pipelines in a single clock cycle. Of course, it will take a number of cycles to refill the pipelines, but as long as the branch prediction logic is accurate, such stalls should be kept to a minimum.

PowerPC 604e

The 604e is an enhanced version of the 604. It's a 2.5 V part made with a 0.35-micron five-layer-metal CMOS process. It has 5.6 million transistors, of which 3.8 million implement the large on-chip caches. The die size is 196 mm^2, smaller than the original 604. At its named clock rate, the 166 MHz 604e dissipates an estimated 10 W. It supports a wide range of processor to bus frequency ratios, which can simplify a system design.

The logic of the load/store unit was beefed up to reduce the number of cycles spent fetching and writing data. Many of the move operations now complete within a single cycle on a cache access. This is accomplished by simultaneously writing the data to both the cache and the requesting execution unit. Additionally, the cache logic forwards a subsequent non-speculative move operation immediately to the load/store unit, rather than waiting for the cache fill to complete as it does on the 100 MHz 604. The 604e has separate code and data caches, each that's 32 KB in size, while the 100 MHz 604 had two separate 16 KB caches. The caches are logically organized as four-way set associative using 256 sets, instead of the 128 sets on the 604. By keeping the cache organization as four-way, this allows the 604e to be pin-compatible with the 604. The processor core operates at 2.5 V, while the bus interface still operates at 3.3V.



Appendix B: Porting to the Power Mac

In this book, you have looked at how to write a Macintosh application so that the C code compiles and runs on both 68K-based Macs and Power Macs. This is fine if you're starting a program from scratch. Of course, the luxury of writing programs this way doesn't exist for vendors with software already on the market. For these folks, the real issue becomes: How hard is it to port existing Mac code to a Power Macintosh? Overall, porting working Mac C code isn't difficult. There will be some problem areas for certain types of applications, which are covered in this appendix.

The program's code should be ANSI C compliant. This is because PowerPC compilers originated from ANSI C compilers. The ANSI C function prototyping is an asset here because it can flag problems with improperly written calls to functions or Toolbox routines.

Some portions of the program might rely on certain compiler dependencies to operate. Obviously, such program elements should be removed. One such dependency is the size of the `int` variable, which can be 16 or 32 bits, depending on a compiler's settings. Eliminate `int` variables from your source code and explicitly declare them as `short` or `long`. If you've ported the code from another platform, most of these dependencies have probably been eliminated. The name `powerc` is defined for use in conditional compilation.

The application code must be well-behaved. That is, it only accesses the hardware through the Toolbox, not by hammering at certain addresses. Also, it must be 32-bit clean. The various hardware configurations that make up the Mac line should have discouraged a misbehaved program, and retooling an application to work with System 7 should have taken care of ensuring the program is 32-bit clean.

The use of low memory globals is strongly discouraged. To this end, the "SysEqu.h" header file has been eliminated. In its place you should use the header file "LowMem.h." Although direct accesses to these areas of memory are still supported (for the moment), you should start using the "accessor functions" in "LowMem.h" to obtain these values. For example, instead of obtaining the value of A5 from the global `CurrentA5` (address 0x904), use the function `LMGetCurrentA5()` and let the Power Mac OS return a value for you.

If you use callback or completion routines, such as those used in the high-level event handlers, custom window controls, or an event filter function in a dialog box, you'll need to build a UPP for the function. This enables the Mixed Mode Manager to deal with your code when it's called by the Macintosh OS. Basically, if the function is accessed using a `ProcPtr`, it better have a UPP set up for it. Fortunately, the PowerPC header files provide macros that handle most of these details for you.

Search for functions prefaced with "New" or "Call" in the header files that you use with the program. If you're writing a custom PowerPC plug-in module to enhance a 680x0 application (as Adobe did with Photoshop 2.5), you'll have to write the UPPs yourself. (See Chapter 5 for details.) If you're writing a PowerPC plug-in module for a PowerPC application, you can use PowerPC procedure pointers and avoid the overhead of a mode switch or the use of UPPs.

If you're passing data structures to the Toolbox, remember that it's mostly emulated 680x0 code and you have to word-align the data for it. Use the compiler declaration `#pragma options align=mac68k` to achieve this. Don't forget to use `#pragma options align=reset` after such structures to provide optimal PowerPC data alignment. If the program and its data are expected to run on 680x0 Macs and Power Macs, you'll need to enforce word-alignment throughout the program. This is also true if you expect to exchange files with 680x0 Macs.

If your program makes heavy use of floating-point math you'll have to make some modifications. The `extended` 80- or 96-bit values, and the 64-bit `comp` used by SANE are not supported in the PowerPC hardware. For compatibility, the PowerPC SANE implementation supports these data types in emulation. To obtain the fastest processing possible, rewrite the code to support the processor's native 32- or 64-bit values. These data types are declared `float` or `double`, respectively.

An 80-bit `long double` type is supported for SANE. Discontinue use of the "sane.h" and "math.h" header files. Instead, use the functions provided in the header file "fp.h," which provides data conversions and transcendental math functions. These functions follow the Floating-Point C Extensions (FPCE) specification, which defines support for IEEE 754/854 floating-point math. As a developing standard, this should enable the program to be ported to other platforms.

Important

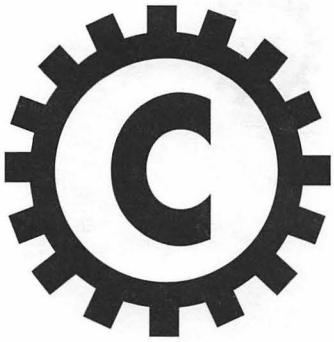
Metrowerks defines the symbol `#pragma IEEEdoubles`. If it is defined (set to 1), the CodeWarrior compiler generates PowerPC 32- and 64-bit values for `float` and `double`. If this name is not defined, the compiler generates 80-bit values that SANE routines use. This allows the same code to be supported on 680x0 Macs and Power Macs, but you might have to rework the code anyway to compensate for the loss in precision if you were relying on 96-bit values.

Be aware that if you've fine tuned the application's processing around the 680x0 environment, you might need to make some readjustments for the PowerPC. The Power Macs use a new Modern Memory Manager that has been optimized for a RISC processor. This memory manager might cause problems with a program that's adapted for the old Memory Manager. Likewise, calling some Toolbox routines can create an ISA context switch. Avoid making Toolbox calls in tight loops. If the loop isn't running as fast as expected, a mode switch is probably occurring.

The `pascal` keyword is ignored by PowerPC compilers. This keyword was used to reorder how a C function's arguments get passed to the target function.

It's primarily used when calling Toolbox functions whose interface was based on the Pascal programming language. This isn't a big issue, because C calling conventions are the norm for the Power Mac software, and the Mixed Mode Manager sorts the rest out for you. However, be aware that Pascal automatically passes arguments larger than 4 bytes by reference, and you'll have to declare such arguments as pointers in C.

Avoid patching traps if you can help it. The Power Macintosh's run-time architecture allows the ready enhancement of applications and other code fragments without resorting to trap patches. If you must patch, take into consideration what the code is doing, versus the overhead of the Mode switch. Write a fat patch if necessary.



Appendix C: Program Listings

Chapter 2

munger.c

```
#include <stdio.h>

#define CR   0x0D
#define LF   0x0A

FILE *istream, *ostream;

void main(void)
{
short   crflag;
long   icount, ocount;
char   ifile[64], ofile[64];  /* Path names must be 64 chars or less */
int      nextbyte;

printf ("Enter input file: ");
gets (ifile);
if ((istream = fopen(ifile, "rb")) == NULL) /* Open the file OK? */
   {
   printf ("\nError opening input\n");   /* NO, say so */
   return;         /* Bail out */
   } /* end if */
```

```
printf ("Enter output file: ");
gets(ofile);
if ((ostream = fopen(ofile, "wb")) == NULL) /* Can we write an output file */
   {
   fclose (istream);            /* NO. First close input file */
   printf ("\nError opening output\n");   /*   then warn, and bail out */
   return;
   } /* end if */

icount = 0L;                  /* Set counters */
ocount = 0L;
crflag = 0;

while((nextbyte = fgetc(istream)) != EOF) /* Read chars until end of file */
   {
   icount++;               /* Bump input char counter */
   switch (nextbyte)            /* What char was read? */
      {
      case CR:
      if (crflag >= 1)         /* Two in a row, end of paragraph */
         {
         fputc(nextbyte, ostream);  /* Write two CRs to the output */
         fputc(nextbyte, ostream);
         crflag = 0;         /* Reset the flag */
         ocount++;
         } /* end if */
      else
         crflag++;         /* Bump the flag, and toss the CR */
      break;
      case LF:            /* Toss LF, but don't touch crflag */
      break;
      default:
         fputc(nextbyte, ostream);  /* All other chars get written */
         ocount++;
         crflag = 0;         /* Clear the flag */
      } /* end switch */
   } /* end while */
fclose (istream);              /* Clean up */
fclose (ostream);
printf("Bytes read:    %ld\n", icount);
printf("Bytes written: %ld\n", ocount);
}
```

process.c

```
#include <processes.h>
#include <memory.h>
#include <strings.h>
#include <stdio.h>

void main (void)
{
register int      i;
ProcessInfoRec   thisProcess;
ProcessSerialNumber   process;
FSSpec         thisFileSpec;
unsigned char      typeBuffer[5] = {0};
unsigned char      signatureBuffer[5] = {0};

thisProcess.processAppSpec = &thisFileSpec;  /* Aim pointer at our storage */
thisProcess.processInfoLength = sizeof(ProcessInfoRec);/* Store record size */
thisProcess.processName = (unsigned char *) NewPtr(32);
/* Allocate room for the name */
process.highLongOfPSN = kNoProcess;   /* Clear out process serial number */
process.lowLongOfPSN = kNoProcess;

while (GetNextProcess(&process) == noErr) /* Loop until all processes found */
   {
   if (GetProcessInformation(&process, &thisProcess) == noErr)
/* Obtain detailed info */
      {
      for (i = 0; i <= 3; i++)  /* Copy type & sig info into string buffers */
         {
         typeBuffer[i] = ((char *) &thisProcess.processType)[i];
         signatureBuffer[i] = ((char *) &thisProcess.processSignature)[i];
         } /* end for */
      printf ("Process SN: %ld, %ld, Type: %s, Signature: %s, Name: ",
         thisProcess.processNumber.highLongOfPSN,
         thisProcess.processNumber.lowLongOfPSN,
         typeBuffer,
         signatureBuffer);
      printf (" %s \n", P2CStr(thisProcess.processName));
/* Now print the name */
      } /* end if */
   } /* end while */
} /* end main() */
```

Chapter 3

hello1.c

```
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Windows.h>
#include <Memory.h>
#include <Events.h>
#include <OSUtils.h>

#define TRUE      true
#define FALSE      false

#define NIL          0L
#define IN_FRONT     (-1)
#define IS_VISIBLE   TRUE
#define NO_CLOSE_BOX FALSE

void main(void)
{
WindowPtr   thisWindow;
Rect        windowRect;

/* Lunge after all the memory we can get */
   MaxApplZone();
   MoreMasters();
   MoreMasters();

/* Initialize the various Managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();

/* Set up the window */
   windowRect.top = windowRect.left = 40;
   windowRect.bottom = 200;
   windowRect.right = 300;
   if ((thisWindow = NewWindow(NIL, &windowRect,
      "\pHello world", IS_VISIBLE, documentProc,
```

```
      (WindowPtr) IN_FRONT, NO_CLOSE_BOX, NIL)) != NIL)
      {
      SetPort(thisWindow); /* Make window current drawing port */
      MoveTo (20, 20);
      DrawString("\pHello world");
      InitCursor();

      while (!Button())  /* Wait until mouse button clicked */
         ;

      DisposeWindow(thisWindow);
      } /* end if */
   else
      SysBeep(30);

} /* end main() */
```

macmunger.c

```
/* Simple app to modify a text file */
/* Copyright © 1994 Tom Thompson, for Hayden   */
/* Creation date:         20-Jan-94 */

#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Controls.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Devices.h>
#include <Memory.h>
#include <Events.h>
#include <Desk.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <ToolUtils.h>
#include <TextUtils.h>
#include <StandardFile.h>
#include <Errors.h>
#include <Resources.h>
#include <DiskInit.h>
```

```
#define LAST_MENU      3   /* Number of menus */
#define APPLE_MENU      128   /* Menu ID for Apple menu */
#define FILE_MENU      129   /* Menu ID for File menu */
#define EDIT_MENU      130   /* Menu ID for Edit menu */
#define RESOURCE_ID      127   /* Starting index into the menu array */

#define ABOUT_BOX      1
/* About box menu item # in Apple menu */

#define OPEN_FILE      1      /* Open item # in File menu */
/*--------------------*/      /* Separator line is item # 2 */
#define I_QUIT         3      /* Quit item # in File menu */

#define ABOUT_BOX_ID   128
/* Resource IDs for our windows & dialogs */
#define STATUS_BOX_ID   129
#define ERROR_BOX_ID   130

/* Various constants */
#define NIL            0L
#define FALSE         false
#define TRUE          true

#define INIT_X         112
/* Coords for disk init dialog box */
#define INIT_Y         80

#define APPEND_MENU      0
#define CHAR_CODE_MASK   255
#define IN_FRONT      -1
#define NO_CURSOR      0L
#define ONE_FILE_TYPE   1
#define LONG_NAP      60L

#define CR            0x0D
#define LF            0x0A

/* Function prototypes */
Boolean Do_Command (long mResult);
Boolean Init_Mac(void);
void Main_Event_Loop(void);
void Report_Error(OSErr errorCode);
```

```
/* Application-specific functions */
void Ask_File(void);
void Munge_File(short input, short output, unsigned char *fileName);

/* Globals */
MenuHandle     gmyMenus[LAST_MENU+1]; /* Handle to our menus */
EventRecord    gmyEvent;  /* Holds the event returned by the OS */
WindowPtr      geventWindow;    /* Our private window */
Boolean         guserDone;
/* Indicates user wants to quit (== TRUE) */
CursHandle      gtheCursor;       /* Current pointer icon */
short         gwindowCode;
WindowPtr      gwhichWindow; /* The window that got an event */

OSType         gfileCreator = {'MUNG'};
/* File type and creator for our output file */
OSType         gfileType = {'TEXT'};

/* Function to report error conditions. Error ID only. */
void Report_Error(OSErr errorCode)
{
unsigned char errNumString[8];

   NumToString((long) errorCode, errNumString);
   ParamText(errNumString, NIL, NIL, NIL);
   StopAlert(ERROR_BOX_ID, NIL);
} /* end Report_Error() */

/* Function to read and write a file. Passed in are the input and output file's */
/*volume */
/*   reference numbers, and the name string of the input file */
void Munge_File(short input, short output, unsigned char *fileName)
{
long         amount;
unsigned char   buffer;
short         crflag;
long          dummyResult;   /* Dummy variable for delay() */
long         icount, ocount;
unsigned char   inNumString[12], outNumString[12];
DialogPtr      statusDialog;

   amount = 1L;
   crflag = 0;
```

```
   icount = 0;
   ocount = 0;
   while (FSRead(input, &amount, &buffer) == noErr)
      {
      icount++;        /* Bump input char counter */
      switch (buffer)      /* What char was read? */
         {
         case CR:
            if (crflag >= 1) /* Two in a row, end of paragraph */
               {
               FSWrite(output, &amount, &buffer);
/* Write two CRs to the output */
               FSWrite(output, &amount, &buffer);
               crflag = 0;        /* Reset the flag */
               ocount++;
               } /* end if */
            else
               crflag++;
/* Bump the flag, and toss the CR */
         break; /* end case CR */
         case LF:   /* Toss LF, but don't touch crflag */
         break;   /* end case LF */
         default:
            FSWrite(output, &amount, &buffer);
            ocount++;
            crflag = 0;               /* Clear the flag */
         } /* end switch */
      } /* end while */

/* Display processing statistics */
   if ((statusDialog = GetNewDialog(STATUS_BOX_ID, NIL, (WindowPtr) IN_FRONT)) !=
   ➥NIL)
      {
      NumToString(icount, inNumString);
/* Convert bytes read to string */
      NumToString(ocount, outNumString);
      ParamText (fileName, inNumString, outNumString, NIL);
      DrawDialog(statusDialog);
      Delay (120L, &dummyResult);
      DisposDialog(statusDialog);
      } /* end if != NIL */
   else
      SysBeep(30);
```

```
} /* end Munge_file() */

/* Obtain info on file to munge and output file */
void Ask_File(void)
{
unsigned char      fileName[14] = {"\pMunge.out"};
short             inFileRefNum, outFileRefNum;
OSErr             fileError;
short             oldVol;
SFTypeList          textType = {'TEXT'};
StandardFileReply   inputReply, outputReply;

/* Open the input file */
   StandardGetFile(NIL, ONE_FILE_TYPE, textType, &inputReply);
   if (inputReply.sfGood)
      {
      GetVol (NIL, &oldVol); /* Save current volume */
      if ((fileError = FSpOpenDF (&inputReply.sfFile, fsCurPerm, &inFileRefNum))
       ➥!= noErr)
         {
         Report_Error(fileError);
         return;
         } /* end if error */

/* Open the output file */
      StandardPutFile ("\pSave text in:", fileName, &outputReply);
      if (outputReply.sfGood)
         {
         SetVol(NIL, outputReply.sfFile.vRefNum);   /* Make the destination volume*/
                                                    /*current */
         fileError = FSpCreate(&outputReply.sfFile, gfileCreator, gfileType,
           ➥smSystemScript);
         switch(fileError)  /* Process result from File Manager */
            {
            case noErr:
            break;
            case dupFNErr: /* File already exists, wipe it out */
              if ((fileError = FSpDelete(&outputReply.sfFile)) == noErr)
                  {
                  if ((fileError = FSpCreate(&outputReply.sfFile, gfileCreator,
                                    gfileType, smSystemScript)) != noErr)
                     {
```

```
                        Report_Error(fileError);
                        FSClose (inFileRefNum);
                        SetVol(NIL, oldVol);
                        return;
                        } /* end if != noErr */
                     } /* end == noErr */
                  else
                     {
                     Report_Error(fileError);
                     FSClose (inFileRefNum);
                     SetVol(NIL, oldVol);
                     return;
                     } /* end else */
               break;   /* end case dupFNErr */
               default:     /* Unknown error, try to abort cleanly */
                  Report_Error(fileError);
                  FSClose (inFileRefNum); /* Close the input file */
                  SetVol(NIL, oldVol);  /* Restore original volume */
                  return;
               } /* end switch */

            if (!(FSpOpenDF (&outputReply.sfFile, fsCurPerm, &outFileRefNum)))
            /* Open data fork */
               {
               gtheCursor = GetCursor(watchCursor);
/* Change the cursor */
               SetCursor(&**gtheCursor);
               Munge_File (inFileRefNum, outFileRefNum, (unsigned char *)
                 ➥inputReply.sfFile.name);
               FSClose (outFileRefNum);
               SetCursor(&qd.arrow);   /* Restore the cursor */
               } /* end if !fileError */
            FlushVol (NIL, outputReply.sfFile.vRefNum);
            } /* end if outputReply.sfGood */
         FSClose (inFileRefNum);
         SetVol(NIL, oldVol);   /* Restore current volume */
         } /* end if inputReply.sfGood */

} /* end Ask_File() */

/* Handle a command thru menu activation. Don't */
/*  forget to unhighlight the selection to indicate */
/* the application is done. (Menu is highlighted */
```

```
/* automagically by MenuSelect.) */

Boolean Do_Command (long mResult)
{
unsigned char   accName[255];
short         itemHit;
Boolean         quitApp;
short         refNum;
DialogPtr      theDialog;
short         theItem, theMenu;
GrafPtr         savePort;
/* place to stow current GrafPort when we activate a */
/* Desk Accessory (DA) */

   quitApp = FALSE;         /* Assume Quit not activated */
   theMenu = HiWord(mResult);   /* Extract the menu selected */
   theItem = LoWord(mResult);   /* Get the item on the menu  */

   switch (theMenu)
      {
      case APPLE_MENU:
         if (theItem == ABOUT_BOX)
/* "About..." selected, describe ourself */
            {
            if ((theDialog = GetNewDialog(ABOUT_BOX_ID, NIL, (WindowPtr)
             ➥IN_FRONT)) != NIL)
               {
               ModalDialog(NIL, &itemHit);
               DisposDialog(theDialog);
               } /* end if != NIL */
            else
               SysBeep(30);
            } /* end if theItem == ABOUT_BOX */
         else                        /* It's a DA */
            {
            GetPort(&savePort);
/* Save port (in case the DA doesn't) */
            GetMenuItemText(gmyMenus[(APPLE_MENU - RESOURCE_ID)], theItem,
             ➥accName);
            refNum = OpenDeskAcc(accName);  /* Start it */
            SetPort(savePort);
/* Done, restore the port */
            }
      break; /* end APPLE_MENU case */
```

```
      case FILE_MENU:
         switch(theItem)
            {
            case OPEN_FILE:
               Ask_File();
               break;
            case I_QUIT:
               quitApp = TRUE;
            break;
            } /* end switch */
      break; /* end FILE_MENU case */

      case EDIT_MENU:
         SystemEdit(theItem - 1);
      break;
      default:
        break;
      } /* end switch */

   HiliteMenu(0);
/* Switch off highlighting on the menu just used */
   return quitApp;
} /* end Do_Command() */

/* The main chunk of code that processes events as they occur. Execution remains */
/* in this loop until Do_Command returns TRUE, indicating the user wants to quit. */
/* In most cases, an event should call a subroutine to handle the event, but in */
/* this example the actions are so simple most code can be placed in-line. */
void Main_Event_Loop(void)
{
Point      where;

   FlushEvents(everyEvent, 0);
/* Clear out left over events */
   guserDone = FALSE;

   do
      {
```

```
    if (WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR))
       {          /* We have an event... */
       switch(gmyEvent.what)   /* Field each type of event */
          {
          case mouseDown:   /* In what window, and where?? */
             gwindowCode = FindWindow(gmyEvent.where, &gwhichWindow);
                switch(gwindowCode)
                {
                case inSysWindow:
 /* It's a Desk Accessory (DA) */
                   SystemClick(&gmyEvent, gwhichWindow);
                break;
                case inDrag:      /* Drag the window */
                break;
                case inGrow:
/* Grow the window, if size has changed */
                break;
                case inContent:
/* Bring window to front if it's not, and that's all */
                break;
                case inMenuBar:
/* In a menu, handle the command */
                   guserDone = Do_Command(MenuSelect(gmyEvent.where));
                break;
                } /* end switch gwindowCode */
             break; /* end mouseDown */
          case keyDown:
          case autoKey:
/* Command key pressed, pass to MenuKey */
             if((gmyEvent.modifiers & cmdKey) != 0)
                guserDone = Do_Command(MenuKey((char) (gmyEvent.message
                                  & CHAR_CODE_MASK)));
              break; /* end key events */
          case updateEvt:            /* Update the window */
                gwhichWindow = (WindowPtr) gmyEvent.message;
              break;
          case diskEvt:   /* Handle disk insertion event */
             if (HiWord(gmyEvent.message) != noErr)
                {
                DILoad();
                where.h = INIT_X;
                where.v = INIT_Y;
                DIBadMount(where, gmyEvent.message);
```

```
                DIUnload();
                } /* end if != noErr */
            break; /* end disk event */
            case activateEvt:         /* Activate event */
                gwhichWindow = (WindowPtr) gmyEvent.message;
            break;
            default:
            break;
            } /* end switch gmyEvent.what */
        } /* end if on next event */
    } /* end do */

  while (guserDone == FALSE);  /* Loop until told to stop */
} /* end Main_Event_Loop() */

Boolean Init_Mac(void)
{
short   i;

/* Lunge after all the memory we can get */
  MaxApplZone();

/* Make sure we've got some master pointers */
  MoreMasters();
  MoreMasters();
  MoreMasters();
  MoreMasters();

/* Initialize managers */
  InitGraf(&qd.thePort);
  InitFonts();
  FlushEvents(everyEvent, 0);
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NIL);

  for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
/* Loop to setup menus */
    {
    gmyMenus[(i - RESOURCE_ID)] = GetMenu(i);
/* Get menu resource */
    if (gmyMenus[(i - RESOURCE_ID)] == NIL)
```

```
/* Didn't get resource? */
        return FALSE;
/* No, sure didn't, bail out */
      }; /* end for */

   AppendResMenu(gmyMenus[(APPLE_MENU - RESOURCE_ID)], 'DRVR');
   /* Build Apple menu */

   for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
/* Add the menus */
      InsertMenu(gmyMenus[(i - RESOURCE_ID)], APPEND_MENU);

   DrawMenuBar();
   InitCursor();    /* Tell user app is ready */
   return TRUE;
} /* end Init_Mac() */

void main(void)
{
   if (Init_Mac())
      Main_Event_Loop();
   else
      SysBeep(30);
} /* end main */
```

SonOMunger.c

```
/* Enhanced app to modify a text file */
/* Copyright © 1994 Tom Thompson, for Hayden   */
/* Creation date:          20-Jan-94 */

#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Controls.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Devices.h>
#include <Memory.h>
#include <Events.h>
#include <Desk.h>
#include <OSEvents.h>
```

```
#include <OSUtils.h>
#include <ToolUtils.h>
#include <TextUtils.h>
#include <StandardFile.h>
#include <Errors.h>
#include <Resources.h>
#include <DiskInit.h>

#include <AppleTalk.h>
#include <AppleEvents.h>
#include <EPPC.h>
#include <PPCToolBox.h>
#include <Processes.h>
#include <LowMem.h>

struct AEinstalls
   {
    AEEventClass theClass;
    AEEventID theEvent;
    AEEventHandlerProcPtr theProc;
   };
typedef struct AEinstalls AEinstalls;

#define LAST_HANDLER   3   /* Number of Apple Event handlers - 1 */
#define LAST_MENU      3      /* Number of menus */

#define APPLE_MENU      128    /* Menu ID for Apple menu */
#define FILE_MENU      129    /* Menu ID for File menu */
#define EDIT_MENU      130    /* Menu ID for Edit menu */
#define RESOURCE_ID      127
/* Starting index into the menu array */

#define ABOUT_BOX      1      /* About box menu item # in Apple menu */

#define OPEN_FILE      1      /* Open item # in File menu */
/*----------------------*/    /* Separator line is item # 2 */
#define I_QUIT         3      /* Quit item # in File menu */

#define ABOUT_BOX_ID   128
/* Resource IDs for our windows & dialogs */
#define STATUS_BOX_ID   129
#define ERROR_BOX_ID   130
#define ERROR_MESS_ID   131
```

```
/* Various constants */
#define NIL             0L
#define FALSE          false
#define TRUE           true

#define INIT_X          112  /* Coords for disk init dialog box */
#define INIT_Y          80

#define APPEND_MENU      0
#define CHAR_CODE_MASK   255
#define DEFAULT_VOL      0
#define IN_FRONT        -1
#define NO_CURSOR        0L
#define ONE_FILE_TYPE    1
#define SHORT_NAP        60L

#define CR              0x0D
#define LF              0x0A

/* Function prototypes */
Boolean Do_Command (long mResult);
Boolean Init_Mac(void);
void Main_Event_Loop(void);
void Report_Error(OSErr errorCode);
void Report_Err_Message(unsigned char *errMess);

/* High-level Apple Event functions */
Boolean Init_AE_Events(void);       /* Install the handlers */
void Do_High_Level(EventRecord *AERecord);
  /* Post high-level event to the dispatch table */
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn); /* Handlers */
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein, AppleEvent *reply,
   ➥long refIn);
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn);
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn);

/* Application-specific functions */
void Ask_File(void);
OSErr Munge_File(short input, short output, unsigned char *fileName);
```

```
/* Globals */
MenuHandle      gmyMenus[LAST_MENU+1];  /* Handle to our menus */
EventRecord     gmyEvent;  /* Holds the event returned by the OS */
WindowPtr       geventWindow;       /* Our private window */
Boolean          guserDone;
/* Indicates user wants to quit (== TRUE) */
CursHandle       gtheCursor;
/* Current pointer icon */
short          gwindowCode;
WindowPtr       gwhichWindow; /* The window that got an event */

OSType          gfileCreator = {'MUNG'};
/* File type and creator for our output file */
OSType          gfileType = {'TEXT'};

void Report_Err_Message(unsigned char *errMess)
{

   ParamText(errMess, NIL, NIL, NIL);
   CautionAlert(ERROR_MESS_ID, NIL);
} /* end Report_Err_Message() */

/* Function to report error conditions. Error ID only. */
void Report_Error(OSErr errorCode)
{
unsigned char errNumString[8];

   NumToString((long) errorCode, errNumString);
   ParamText(errNumString, NIL, NIL, NIL);
   StopAlert(ERROR_BOX_ID, NIL);
} /* end Report_Error() */

/* Function to read and write a file. Passed in are the input and output file's */
/* volume */
/*   reference numbers, and the name string of the input file */
OSErr Munge_File(short input, short output, unsigned char *fileName)
{
long          amount;
unsigned char   buffer;
short          crflag;
long          dummyResult;
EventRecord       dummyBuffer;
OSErr          fInOutErr;
```

```
long         icount, ocount;
unsigned char   inNumString[12], outNumString[12];
short         nextTime, startTime;
DialogPtr      statusDialog;

   amount = 1L;
   crflag = 0;
   icount = 0;
   ocount = 0;
   nextTime = 0;
   startTime = LMGetTicks();

   while (FSRead(input, &amount, &buffer) == noErr)
      {
      icount++;          /* Bump input char counter */
      switch (buffer)      /* What char was read? */
         {
         case CR:
            if (crflag >= 1) /* Two in a row, end of paragraph */
               {
               if (!(fInOutErr = FSWrite(output, &amount, &buffer)))
                  {
                  if ((fInOutErr = FSWrite(output, &amount, &buffer)) != noErr)
                     {
                     Report_Error(fInOutErr);
                     return fInOutErr;
                     } /* end if != */
                  } /* end if ! */
               else
                  {
                  Report_Error(fInOutErr);
                  return fInOutErr;
                  } /* end else */
               crflag = 0;            /* Reset the flag */
               ocount++;
               } /* end if */
            else
               crflag++;
/* Bump the flag, and toss the CR */
         break; /* end case CR */
         case LF:  /* Toss LF, but don't touch crflag */
         break;   /* end case LF */
         default:                /* Write a character out */
```

```
            if ((fInOutErr = FSWrite(output, &amount, &buffer)) != noErr)
               {
               Report_Error(fInOutErr);
               return fInOutErr;
               } /* end if */
            ocount++;
            crflag = 0;              /* Clear the flag */
         break;
         } /* end switch */
      } /* end while */

/* Display processing statistics */
   if ((statusDialog = GetNewDialog(STATUS_BOX_ID, NIL, (WindowPtr) IN_FRONT)) !=
     ➥NIL)
      {
      NumToString(icount, inNumString);
/* Convert bytes read to string */
      NumToString(ocount, outNumString);
      ParamText (fileName, inNumString, outNumString, NIL);
      DrawDialog(statusDialog);
      Delay (120L, &dummyResult);
      DisposDialog(statusDialog);
      } /* end if != NIL */
   else
      SysBeep(30);

   return fInOutErr;
} /* end Munge_file() */

/* Obtain info on file to munge and output file */
void Ask_File(void)
{
unsigned char      fileName[14] = {"\pMunge.out"};
short              inFileRefNum, outFileRefNum;
OSErr              fileError;
short              oldVol;
SFTypeList         textType = {'TEXT'};
StandardFileReply   inputReply, outputReply;

/* Open the input file */
   StandardGetFile(NIL, ONE_FILE_TYPE, textType, &inputReply);
   if (inputReply.sfGood)
      {
```

```
      GetVol (NIL, &oldVol);            /* Save current volume */
      if ((fileError = FSpOpenDF (&inputReply.sfFile, fsCurPerm, &inFileRefNum))
        ➥!= noErr)
         {
         Report_Error(fileError);
         return;
         } /* end if error */

/* Open the output file */
      StandardPutFile ("\pSave text in:", fileName, &outputReply);
      if (outputReply.sfGood)
         {
         SetVol(NIL, outputReply.sfFile.vRefNum);
/* Make the destination volume current */
         fileError = FSpCreate(&outputReply.sfFile, gfileCreator, gfileType,
           ➥smSystemScript);
         switch(fileError)
/* Process result from File Manager */
            {
            case noErr:
            break;
            case dupFNErr:
/* File already exists, wipe it out */
               if ((fileError = FSpDelete(&outputReply.sfFile)) == noErr)
                  {
                  if ((fileError = FSpCreate(&outputReply.sfFile, gfileCreator,
                                    gfileType, smSystemScript)) != noErr)
                     {
                     Report_Error(fileError);
                     FSClose (inFileRefNum);
                     SetVol(NIL, oldVol);
                     return;
                     } /* end if != noErr */
                  } /* end == noErr */
               else
                  {
                  Report_Error(fileError);
                  FSClose (inFileRefNum);
                  SetVol(NIL, oldVol);
                  return;
                  } /* end else */
            break;   /* end case dupFNErr */
            default:  /* Unknown error, try to abort cleanly */
```

```
                Report_Error(fileError);
                FSClose (inFileRefNum);
 /* Close the input file */
                SetVol(NIL, oldVol);  /* Restore original volume */
                return;
              } /* end switch */

          if (!(FSpOpenDF (&outputReply.sfFile, fsCurPerm, &outFileRefNum))) /*
Open data fork */
              {
              gtheCursor = GetCursor(watchCursor);
/* Change the cursor */
              SetCursor(&**gtheCursor);
              Munge_File (inFileRefNum, outFileRefNum, (unsigned char *)
                ➥inputReply.sfFile.name);
              FSClose (outFileRefNum);
              SetCursor(&qd.arrow); /* Restore the cursor */
              } /* end if !fileError */
           FlushVol (NIL, outputReply.sfFile.vRefNum);
           } /* end if outputReply.sfGood */
        FSClose (inFileRefNum);
        SetVol(NIL, oldVol);  /* Restore current volume */
        } /* end if inputReply.sfGood */

} /* end Ask_File() */

Boolean Init_AE_Events(void)
{
OSErr   err;
short   i;
static AEinstalls HandlersToInstall[] =
/* The 4 required Apple Events */
   {
      {kCoreEventClass, kAEOpenApplication, (AEEventHandlerProcPtr)
          ➥Core_AE_Open_Handler},
      {kCoreEventClass, kAEOpenDocuments, (AEEventHandlerProcPtr)
          ➥Core_AE_OpenDoc_Handler},
      {kCoreEventClass, kAEQuitApplication, (AEEventHandlerProcPtr)
          ➥Core_AE_Quit_Handler},
      {kCoreEventClass, kAEPrintDocuments, (AEEventHandlerProcPtr)
          ➥Core_AE_Print_Handler}
    };

    for (i = 0; i < LAST_HANDLER; i++)
```

```
      {
      err = AEInstallEventHandler(HandlersToInstall[i].theClass,
HandlersToInstall[i].theEvent,
                           NewAEEventHandlerProc(HandlersToInstall[i].theProc), 0, FALSE);
      if (err)      /* If there was a problem, bail out */
         return FALSE;
      } /* end for */

   return TRUE;
} /* end Init_AE_Events() */

/* High-level open application event.  */
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein,
    ➥AppleEvent *reply, long refIn)
{
    return noErr;
} /* end Core_AE_Open_Handler() */

/* High-level open document event */
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein,
    ➥AppleEvent *reply, long refIn)
{
short         i, j;
AEDesc         fileDesc;
OSErr         highLevelErr;
AEKeyword       ignoredKeyWord;
DescType       ignoredType;
Size         ignoredSize;
long         numberOFiles;
unsigned char   outFileName[64];
FSSpec         inFSS, outFSS;
short         inFileRefNum, outFileRefNum;
OSErr         fInErr, fOutErr, mungeResult;


   gtheCursor = GetCursor(watchCursor);
/* Change the cursor to indicate we're busy */
   SetCursor(&**gtheCursor);
   mungeResult = 0;  /* Clear result so for loop will operate */
/* Get parameter info (a list of file names) out of Apple Event*/
   if (!(highLevelErr = AEGetParamDesc(messagein, keyDirectObject, typeAEList,
       ➥&fileDesc)))
      {
```

```
     if ((highLevelErr = AECountItems(&fileDesc, &numberOFiles)) == noErr)
/* Count files */
        {
        for (i = 1; ((i <= numberOFiles) && (!highLevelErr) && (!mungeResult)); ++i)
           {
           if (!(highLevelErr = AEGetNthPtr(&fileDesc, i, typeFSS,
&ignoredKeyWord, &ignoredType,
                                 (char *)&inFSS, sizeof(inFSS), &ignoredSize)))
/* Get each name */
              {
              for (j = 1; (j <= inFSS.name[0]); j++)
/* Copy input file name to file output name */
                 {
                 outFileName[j] = inFSS.name[j];
                 } /* end for */
              outFileName[j] = '.';
/* Tack on a '.out' extension */
              outFileName[j + 1] = 'o';
              outFileName[j + 2] = 'u';
              outFileName[j + 3] = 't';
              outFileName[0] = (j + 3);
/* Update the string's length */
              if (!(fInErr = FSpOpenDF(&inFSS, fsCurPerm, &inFileRefNum)))
                 {
                 if ((fOutErr = FSMakeFSSpec(DEFAULT_VOL, NIL, outFileName,
                   ➥&outFSS)) == fnfErr)
                    {
                    if (!(fOutErr = FSpCreate(&outFSS, gfileCreator, gfileType,
                     ➥smSystemScript)))
                       {
                       if (!(fOutErr = FSpOpenDF(&outFSS, fsCurPerm,
                        ➥&outFileRefNum)))
                          {
                          mungeResult = Munge_File(inFileRefNum, outFileRefNum,
                            ➥inFSS.name);  /* Process the data */
                          FlushVol(NIL, outFileRefNum);
                          FSClose(outFileRefNum);
                          } /* end if !fOutErr */
                       else
                          Report_Err_Message("\pError opening output file");
                       } /* end if !fOutErr */
                    else
                       {
                       Report_Err_Message("\pError creating output file");
```

```
                        } /* end else */
                     } /* end if == fnfErr */
                  else
                     {
                     if (fOutErr == noErr)
/* No error means a file already has that name */
                        Report_Err_Message("\pCan't write, file already exists");
                     } /* end else */
                  FSClose(inFileRefNum);
                  } /* end if !fInErr */
               else
                  Report_Err_Message("\pError opening input file");
               } /* end if !highlevelErr */
            } /* end for */
         } /* end if == noErr */
         highLevelErr = AEDisposeDesc(&fileDesc);
/* Dispose of the copy made by AEGetParamDesc() */
      } /* end if !highLevelErr */

   SetCursor(&qd.arrow);          /* Restore the cursor */
   guserDone = TRUE;             /* We're done, stop the application */
   return highLevelErr;
} /* end Core_AE_OpenDoc_Handler() */

/* High-level print event */
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein,
   ➥AppleEvent *reply, long refIn)
{
    return errAEEventNotHandled;
/* No printing done here, so no print handler */
} /* end Core_AE_Print_Handler() */

/* High-level quit event  */
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein,
   ➥AppleEvent *reply, long refIn)
{
    guserDone = TRUE;
/* Tell main event loop we want to stop */
    return noErr;
} /* Core_AE_Quit_Handler() */

void Do_High_Level(EventRecord *AERecord)
{
```

```
   AEProcessAppleEvent(AERecord);
} /* end Do_High_Level() */

/* Handle a command thru menu activation. Don't forget to unhighlight the
   selection to indicate the application is done. (Menu is highlighted
   automagically by MenuSelect.) */

Boolean Do_Command (long mResult)
{
unsigned char   accName[255];
short         itemHit;
Boolean         quitApp;
short         refNum;
DialogPtr      theDialog;
short         theItem, theMenu;
GrafPtr         savePort;      /* place to stow current GrafPort when we
                           activate a Desk Accessory (DA) */

   quitApp = FALSE;          /* Assume Quit not activated */
   theMenu = HiWord(mResult);   /* Extract the menu selected */
   theItem = LoWord(mResult);   /* Get the item on the menu  */

   switch (theMenu)
      {
      case APPLE_MENU:
         if (theItem == ABOUT_BOX)
/* "About..." selected, describe ourself */
            {
            if ((theDialog = GetNewDialog(ABOUT_BOX_ID, NIL, (WindowPtr)
              ➥IN_FRONT)) != NIL)
               {
               ModalDialog(NIL, &itemHit);
               DisposDialog(theDialog);
               } /* end if != NIL */
            else
               SysBeep(30);
            } /* end if theItem == ABOUT_BOX */
         else        /* It's a DA */
            {
            GetPort(&savePort);   /* Save port (in case the DA doesn't) */
            GetMenuItemText(gmyMenus[(APPLE_MENU - RESOURCE_ID)], theItem,
               ➥accName);
            refNum = OpenDeskAcc(accName);  /* Start it */
```

```
            SetPort(savePort); /* Done, restore the port */
            }
      break; /* end APPLE_MENU case */

      case FILE_MENU:
         switch(theItem)
            {
            case OPEN_FILE:
               Ask_File();
               break;
            case I_QUIT:
               quitApp = TRUE;
            break;
            } /* end switch */
      break; /* end FILE_MENU case */

      case EDIT_MENU:
         SystemEdit(theItem - 1);
      break;
      default:
        break;
      } /* end switch */

   HiliteMenu(0);
/* Switch off highlighting on the menu just used */
   return quitApp;
} /* end Do_Command() */

/* The main chunk of code that processes events as they occur. Execution remains*/
/* in this loop until Do_Command returns TRUE, indicating the user wants to quit.*/
/* In most cases, an event should call a subroutine to handle the event, but in*/
/* this example the actions are so simple most code can be placed in-line. */
void Main_Event_Loop(void)
{
Point      where;

   FlushEvents(everyEvent, 0);  /* Clear out left over events */
   guserDone = FALSE;
```

```
   do
      {
      if (WaitNextEvent(everyEvent, &gmyEvent, SHORT_NAP, NO_CURSOR))
         {                    /* We have an event... */
         switch(gmyEvent.what)    /* Field each type of event */
            {
            case mouseDown:   /* In what window, and where?? */
               gwindowCode = FindWindow(gmyEvent.where, &gwhichWindow);
                  switch(gwindowCode)
                  {
                  case inSysWindow:   /* It's a Desk Accessory (DA) */
                     SystemClick(&gmyEvent, gwhichWindow);
                  break;
                  case inDrag:      /* Drag the window */
                  break;
                  case inGrow:
/* Grow the window, if size has changed */
                  break;
                  case inContent:
/* Bring window to front if it's not, and that's all */
                  break;
                  case inMenuBar:
/* In a menu, handle the command */
                     guserDone = Do_Command(MenuSelect(gmyEvent.where));
                  break;
                  } /* end switch gwindowCode */
               break; /* end mouseDown */
            case keyDown:
            case autoKey:
/* Command key pressed, pass to MenuKey */
               if((gmyEvent.modifiers & cmdKey) != 0)
                  guserDone = Do_Command(MenuKey((char) (gmyEvent.message
                                 & CHAR_CODE_MASK)));
                break; /* end key events */
            case updateEvt:            /* Update the window */
                  gwhichWindow = (WindowPtr) gmyEvent.message;
               break;
            case diskEvt:
/* Handle disk insertion event */
               if (HiWord(gmyEvent.message) != noErr)
                  {
                  DILoad();
                  where.h = INIT_X;
```

```
                where.v = INIT_Y;
                DIBadMount(where, gmyEvent.message);
                DIUnload();
                } /* end if != noErr */
            break; /* end disk event */
            case activateEvt:         /* Activate event */
                  gwhichWindow = (WindowPtr) gmyEvent.message;
            break;
            case kHighLevelEvent:      /* Handle Apple Event */
               Do_High_Level(&gmyEvent);
            break;
            default:
            break;
            } /* end switch gmyEvent.what */
         } /* end if on next event */
      } /* end do */

   while (guserDone == FALSE);   /* Loop until told to stop */
} /* end Main_Event_Loop() */

Boolean Init_Mac(void)
{
short   i;

/* Lunge after all the memory we can get */
   MaxApplZone();

/* Make sure we've got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
```

```
   TEInit();
   InitDialogs(NIL);

   for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
/* Loop to setup menus */
      {
      gmyMenus[(i - RESOURCE_ID)] = GetMenu(i);
/* Get menu resource */
      if (gmyMenus[(i - RESOURCE_ID)] == NIL)
/* Didn't get resource? */
         return FALSE;   /* No, sure didn't, bail out */
      }; /* end for */

        AppendResMenu(gmyMenus[(APPLE_MENU - RESOURCE_ID)], 'DRVR');
        /* Build Apple menu */

   for (i = APPLE_MENU; i < (APPLE_MENU + LAST_MENU); i++)
/* Add the menus */
      InsertMenu(gmyMenus[(i - RESOURCE_ID)], APPEND_MENU);

   DrawMenuBar();

   if (!Init_AE_Events())
 /* Set up our high-level event handlers */
      return FALSE;

   InitCursor();  /* Tell user app is ready */
   return TRUE;
} /* end Init_Mac() */

void main(void)
{
   if (Init_Mac())
      Main_Event_Loop();
   else
      SysBeep(30);
} /* end main */
```

Chapter 5

SwitchBank.c

```
/* SwitchBank - Apple Event application that can eject "captive*/
/*            volumes". (The volume, usually a CD, can't be*/
/*             ejected because File Sharing (FS) is on.) */
/*                                                        */
/* Creation date:     23-Jan-94 */
/*   Added server call to halt FS, instead of using a */
/*      Quit Apple Event (I'm told this is the safer*/
/*      way to do this.)             26-Jan-94 */
/*   Changed code to look at volume to see if it's shared,*/
/*      rather than just snoop for the File Sharing*/
/*      Extension. This way we can eject other volumes*/
/*      without restarting FS frequently, thereby*/
/*      fragmenting the heap. This also lets us eject*/
/*      a volume as FS starts up, without interfering  30-Jan-94*/
/*      with that operation.  */
/* Changed code to use FindFolder() to locate startup 24-Feb-94 */
/*  volume. Also moved error messages into strings.*/
/*      Fixed a bug where the program wasn't releasing*/
/*      the memory used by AEGetParamDesc(). */

#include <Types.h>
#include <ConditionalMacros.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Controls.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Devices.h>
#include <Memory.h>
#include <Files.h>
#include <Events.h>
#include <Desk.h>
#include <OSEvents.h>
#include <ToolUtils.h>
#include <DiskInit.h>
#include <Folders.h>
```

```
#include <AppleTalk.h>
#include <AppleEvents.h>
#include <EPPC.h>
#include <PPCToolBox.h>
#include <Processes.h>

/* Definitions */
#define LAST_MENU            4     /* Number of menus */
#define LAST_HANDLER         3
 /* Number of Apple Event handlers - 1 */

#define MENU_BAR_ID            128 /* ID for MBAR resource */
#define APPLE_MENU             128 /* Menu ID for Apple menu */
#define FILE_MENU              129  /* Menu ID for File menu */
#define EDIT_MENU              130  /* Menu ID for Edit menu */
#define SWITCH_MENU              131
/* Menu ID for File Share control */
#define RESOURCE_ID              127
/* Starting index into the menu array */

#define ABOUT_BOX              1
 /* About box menu item # in Apple menu */

#define I_QUIT                 1   /* Quit item # in File menu */

/* Various constants */
#define NIL                    0L
#define FALSE                 false
#define TRUE                 true

#define INIT_X                 112
/* Coords for disk init dialog box */
#define INIT_Y                 80

#define APPEND_MENU              0
#define CHAR_CODE_MASK           255
#define DEFAULT_VOL              0
#define IN_FRONT              (-1)
#define MAX_TRIES              6
#define NO_CURSOR              0L
#define LONG_NAP              60L
#define SYSTEM_7              0x0700
#define FILE_SHARING_CREATOR   'hhgg'
```

```
#define FILE_SHARING_TYPE      'INIT'

#define ABOUT_BOX_ID         128
/* Resource IDs for our windows & dialogs */
#define ERROR_BOX_ID         130
#define ERROR_MESS_ID         131

#define LOG_ID_STR            128/* Resource ID for the message strings */
#define PROBLEM_STOPPING_FS      1 /* ID numbers of the messages */
#define PROBLEM_STARTING_FS      2
#define PROBLEM_ON_EJECT      3
#define DONT_EJECT_STARTUP_VOL   4
#define CANT_FIND_STARTUP_VOL   5
#define TROUBLE_WITH_SYS_INFO   6
#define CANT_LOCATE_FILE      7
#define PROBLEM_WITH_AE_HANDLER   8
#define SYSTEM_7_REQUIRED      9

#define PERSONAL_ACCESS_MASK   0x00000200L
/* Bit 9 in vMAttrib field = volume is shared */
#define SEND_MESSAGE         13
/* Send a message to the file server */
#define SHUT_DOWN            2
/* csCode to shut down the server */

/* Function prototypes */
Boolean Check_System(void);   /* Standard application functions */
Boolean Do_Command (long mResult);
void Main_Event_Loop(void);
Boolean Init_Mac(void);
void Report_Error(OSErr errorCode);
void Report_Err_Message(long messageID);

Boolean Init_AE_Events(void);   /* High level Apple Events */
void Do_High_Level(EventRecord *AERecord);
pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn); /* Handlers */
pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein, AppleEvent *reply,
   ➥long refIn);
pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn);
pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein, AppleEvent *reply, long
   ➥refIn);
```

```
Boolean File_Share_On(short vRefNum);
  /* Functions to handle details of file sharing */
void Stop_File_Sharing(void);
void Start_File_Sharing(void);
void Toggle_File_Sharing(void);
Boolean Get_FS_Info(void);
Boolean Find_File_Sharing(void);

/* Assorted structures for server trap */
typedef long   *LongIntPtr;

#if defined(powerc) ¦¦ defined (__powerc)
#pragma options align=mac68k
#endif

struct DisconnectParam
   {
   QElemPtr   qLink;
    short      qType;
    short      ioTrap;
    Ptr         ioCmdAddr;
    ProcPtr      ioCompletion;
    OSErr      ioResult;
   LongIntPtr   scDiscArrayPtr;
   short      scArrayCount;
   short      reserved;
   short      scCode;
   short      scNumMinutes;
   short      scFlags;
   StringPtr   scMessagePtr;
   };

#if defined(powerc) ¦¦ defined(__powerc)
#pragma options align=reset
#endif

typedef struct DisconnectParam DisconnectParam;
typedef union SCParamBlockRec SCParamBlockRec;
typedef SCParamBlockRec *SCParamBlockPtr;

/*Structure for installing handlers into AE event dispatch table*/
struct AEinstalls
   {
```

```
    AEEventClass theClass;
    AEEventID theEvent;
    AEEventHandlerProcPtr theProc;
   };
typedef struct AEinstalls AEinstalls;

/* Globals - standard */
WindowPtr         geventWindow;     /* our private window */
EventRecord       gmyEvent;
CursHandle         gtheCursor;    /* Current pointer icon */
Boolean            guserDone;
WindowPtr         gwhichWindow;
short            gwindowCode;

/* Application-specific globals */
short            gdragNDropFlag;
ProcessInfoRec      gprocess;
ProcessSerialNumber   gprocessSN;
long            gSysDirID;
short            gsysVRefNum;
FSSpec            gthisFileSpec;
FSSpecPtr         gthisFileSpecPtr;


void Report_Err_Message(long messageID)
{
unsigned char errorString[256];

   GetIndString((unsigned char *) errorString, LOG_ID_STR, messageID);
   if (errorString[0] == 0)  /* Is there a string present? */
      {
      SysBeep(30);        /* No, give up */
      return;
      } /* end if */
   ParamText(errorString, NIL, NIL, NIL);
   CautionAlert(ERROR_MESS_ID, NIL);

} /* end Report_Err_Message() */

void Report_Error(OSErr errorCode)
{
unsigned char errNumString[8];
```

```
   NumToString((long) errorCode, errNumString);
   ParamText(errNumString, NIL, NIL, NIL);
   StopAlert(ERROR_BOX_ID, NIL);
} /* end Report_Error() */

/* Look for File Sharing Extension process in memory. Do search by signature */
/*   creator & type rather than by file name, so that code works overseas. */
Boolean Get_FS_Info(void)
{
   gthisFileSpecPtr = &gthisFileSpec;
   gprocessSN.highLongOfPSN = kNoProcess;
   gprocessSN.lowLongOfPSN = kNoProcess;

   gprocess.processInfoLength = sizeof(ProcessInfoRec);
/* Store size of record */
   gprocess.processAppSpec = gthisFileSpecPtr;
/* Direct towards our storage */

   while (GetNextProcess(&gprocessSN) == noErr)
/* Loop until all processes found */
      {
      if (GetProcessInformation(&gprocessSN, &gprocess) == noErr)
        /* Obtain detailed info */
         {
         if (gprocess.processType == FILE_SHARING_TYPE &&
/* Is the process File */
             gprocess.processSignature == FILE_SHARING_CREATOR)
/*    Sharing Extension? */
             return TRUE;
         } /* end if */
      } /* end while */

   return FALSE;
}/* end Get_FS_Info() */

/* Determine if the volume in question is being shared.*/
/* If it is, save the File */
/*   Sharing process info so that we can restart it later. */
Boolean File_Share_On(short volRefNum)
{
HParamBlockRec         ioHPB, volHPB;
GetVolParmsInfoBuffer   volInfoBuffer;
```

```
/* Get volume reference number */
   volHPB.volumeParam.ioCompletion = NIL;
/* No completion routine */
   volHPB.volumeParam.ioNamePtr = NIL;  /* No volume name */
   volHPB.volumeParam.ioVRefNum = volRefNum;
   volHPB.volumeParam.ioVolIndex = 0;
/* 0 = Use only volRefNum to obtain the info */
   if (!PBHGetVInfo(&volHPB, FALSE))
      {
/* Get volume's characteristics */
      ioHPB.ioParam.ioCompletion = NIL;
      ioHPB.ioParam.ioNamePtr = NIL;
      ioHPB.ioParam.ioVRefNum = volHPB.volumeParam.ioVRefNum;
/* from PBHGetVInfo() */
      ioHPB.ioParam.ioBuffer = (char *) &volInfoBuffer;
      ioHPB.ioParam.ioReqCount = sizeof(volInfoBuffer);
      if (!PBHGetVolParms(&ioHPB, FALSE))
         {
         if (volInfoBuffer.vMAttrib & PERSONAL_ACCESS_MASK)
/* The disk is shared */
            {
            if (Get_FS_Info())  /* Look for the File Sharing Extension */
               return TRUE;
  /* Got the file info we need to restart sharing */
            } /* end if */
         } /* end if !PBHGetVolParms */
      } /* end if !PBHGetVInfo */
   return FALSE;
} /* end File_Share_On() */


/* Launch the file that has the File Sharing application in it. The file name used
*/ for the launch was obtained from the process when it's memory, or by */
*/searching the start up disk */
void Start_File_Sharing(void)
{
OSErr            launchErr;
LaunchPBPtr         thisAppPBPtr;
LaunchParamBlockRec   thisAppParams;

   gthisFileSpecPtr = &gthisFileSpec;
   thisAppPBPtr = &thisAppParams;
   thisAppParams.launchBlockID = extendedBlock;
```

```
/* Use the new format */
   thisAppParams.launchEPBLength = extendedBlockLen;
   thisAppParams.launchFileFlags = 0;
/* Don't care about file flags */
   thisAppParams.launchControlFlags = (launchNoFileFlags + launchContinue +
      ➥launchDontSwitch);
   thisAppParams.launchAppSpec = gthisFileSpecPtr;
/* Give it file name grabbed */
/*  by Get_FS_Info() before File */
/*   Sharing was stopped */
   thisAppParams.launchAppParameters = NIL;
/* Send just Open event */

   if ((launchErr = LaunchApplication(thisAppPBPtr)) == noErr)
      WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
   else
      Report_Err_Message(PROBLEM_STARTING_FS);

} /* end Start_File_Sharing() */

/* Look for the File Sharing Extension file. User might not have started File */
/* Sharing yet, so we can't grab the name from a process that isn't there. So, we */
/* search the boot disk. */
Boolean Find_File_Sharing(void)
{
HParamBlockRec   searchPB;
FInfo         fileSharingExtInfo, fileSharingMaskInfo;
CInfoPBRec      searchSpec1, searchSpec2;
Point         nilPoint = {0, 0};

/* Set up creator and type for File Sharing Extension */
   fileSharingExtInfo.fdType = FILE_SHARING_TYPE;
   fileSharingExtInfo.fdCreator = FILE_SHARING_CREATOR;
   fileSharingExtInfo.fdFlags = 0;
   fileSharingExtInfo.fdLocation = nilPoint;
   fileSharingExtInfo.fdFldr = 0;

/* Set up masks */
   fileSharingMaskInfo.fdType = (OSType) 0xffffffff;
   fileSharingMaskInfo.fdCreator = (OSType) 0xffffffff;
   fileSharingMaskInfo.fdFlags = 0;
```

```
   fileSharingMaskInfo.fdLocation = nilPoint;
   fileSharingMaskInfo.fdFldr = 0;

/* 1st spec block */
   searchSpec1.hFileInfo.ioNamePtr = NIL;
/* Search by file type, not name */
   searchSpec1.hFileInfo.ioFlFndrInfo = fileSharingExtInfo;
/* Type & creator to look for */

/* 2nd spec block */
   searchSpec2.hFileInfo.ioNamePtr = NIL;
   searchSpec2.hFileInfo.ioFlFndrInfo = fileSharingMaskInfo;
/* Mask */

/* Set up search call */
   searchPB.csParam.ioCompletion = NIL;
   searchPB.csParam.ioNamePtr = NIL;
/* No volume name */
   searchPB.csParam.ioVRefNum = gsysVRefNum;
/* Search on startup volume */
   searchPB.csParam.ioMatchPtr = &gthisFileSpec;
/* Search result goes here */
   searchPB.csParam.ioReqMatchCount = 1;
/* Looking for 1 file */
   searchPB.csParam.ioSearchBits = fsSBFlFndrInfo;
/* Search based on file characteristics */
   searchPB.csParam.ioSearchInfo1 = &searchSpec1;
   searchPB.csParam.ioSearchInfo2 = &searchSpec2;
   searchPB.csParam.ioSearchTime = 0; /* Don't time out */
   searchPB.csParam.ioCatPosition.initialize = 0;
/* Start at the begining */
   searchPB.csParam.ioOptBuffer = NIL;
/* No search cache required */
   searchPB.csParam.ioOptBufSize = 0;

   if (PBCatSearchSync((CSParamPtr) &searchPB) == noErr)
      return TRUE;
   else
      {
      Report_Err_Message(CANT_LOCATE_FILE);
      return FALSE;
      } /* end else */
```

```
} /* end Find_File_Sharing() */

void Toggle_File_Sharing(void)
{
   if (Get_FS_Info())
/* File Sharing already on (and in memory)? */
      Stop_File_Sharing();     /* Yes, turn it off */
   else          /* No, look for the file */
      {
      if (Find_File_Sharing()) /* Find the File Sharing Extension file */
         Start_File_Sharing();    /* Launch it */
      } /* end else */

} /* end Toggle_File_Sharing() */

/* Build high-level event dispatch table and add our handlers to it. Must use*/
/* static declaration so that the dispatch table has file scope. */
Boolean Init_AE_Events(void)
{
OSErr   err;
short   i;
static AEinstalls HandlersToInstall[] =     /* The 4 required Apple Events */
   {
      {kCoreEventClass, kAEOpenApplication, (AEEventHandlerProcPtr)
Core_AE_Open_Handler},
      {kCoreEventClass, kAEOpenDocuments, (AEEventHandlerProcPtr)
Core_AE_OpenDoc_Handler},
      {kCoreEventClass, kAEQuitApplication, (AEEventHandlerProcPtr)
Core_AE_Quit_Handler},
      {kCoreEventClass, kAEPrintDocuments, (AEEventHandlerProcPtr)
Core_AE_Print_Handler},
    };

    for (i = 0; i < LAST_HANDLER; i++)
      {      /* Install each handler in application dispatch table, with a routine*/
             /* descriptor */
      err = AEInstallEventHandler(HandlersToInstall[i].theClass,
HandlersToInstall[i].theEvent,
                          NewAEEventHandlerProc(HandlersToInstall[i].theProc), 0,
                             ➥FALSE);
      if (err)      /* If there was a problem, bail out */
         {
         Report_Err_Message (PROBLEM_WITH_AE_HANDLER);
```

```
        return FALSE;
        } /* end if */
      } /* end for */

   return TRUE;
} /* end Init_AE_Events() */

/* High-level open application event.  */
   pascal OSErr Core_AE_Open_Handler(AppleEvent *messagein, AppleEvent *reply,
     ➥long refIn)
{
    return noErr;
} /* end Core_AE_Open_Handler() */

/* High-level open document event */
   pascal OSErr Core_AE_OpenDoc_Handler(AppleEvent *messagein, AppleEvent *reply,
     ➥long refIn)
{
long          dummyResult;         /* Dummy variable for delay() */
register short   i, j;
Boolean         fileShareWasOn;
AEDesc         volDesc;            /* Container for sent volume names */
OSErr         volErr, highLevelErr;
long         numberOVolumes;         /* Number of volumes dropped onto us */
AEKeyword       ignoredKeyWord;
 /* Bit buckets for high-level event info we don't need */
DescType       ignoredType;
Size         ignoredSize;
FSSpec         volFSS;   /* Container for volume names as FSSPecs */

   gtheCursor = GetCursor(watchCursor);
    /* Change the cursor to indicate we're busy */
   SetCursor(&**gtheCursor);
   fileShareWasOn = FALSE;  /* Assume File Sharing on */

   if (!(highLevelErr = AEGetParamDesc(messagein, keyDirectObject, typeAEList,
      ➥&volDesc)))
      {
      if ((highLevelErr = AECountItems(&volDesc, &numberOVolumes)) == noErr)
/* How many? */
        {
        for (i = 1; ((i <= numberOVolumes) && (!highLevelErr)); ++i)
/* Process each vol */
          {
```

```
            if (!(highLevelErr = AEGetNthPtr(&volDesc, i, typeFSS,
               ➥&ignoredKeyWord, &ignoredType,
                                    (char *)&volFSS, sizeof(volFSS),
                                       ➥&ignoredSize)))
               {
               if (volFSS.vRefNum != gsysVRefNum)
/* Chosen volume the boot drive? */
                  {
                  if (File_Share_On(volFSS.vRefNum))
/* This volume being shared? */
                     {
                     Stop_File_Sharing();
/* Yes, turn it off, set flag */
                     fileShareWasOn = TRUE;
                     } /* end if */
                  j = 0;   /* Set retry count */
                  while (((volErr = Eject(volFSS.name,volFSS.vRefNum)) != noErr) &&
                        (j < MAX_TRIES))
                     {
                     WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
                     Delay(10L, &dummyResult);
                     j++;
                     } /* end while */
                  if (volErr == noErr)            /* Volume ejected OK? */
                     UnmountVol(volFSS.name, volFSS.vRefNum);
                  else
                     Report_Err_Message(PROBLEM_ON_EJECT);
                  } /* end if != gsysVRefNum */
               else
                  Report_Err_Message(DONT_EJECT_STARTUP_VOL);
               } /* end if !highLevelErr */
            } /* end for */
         } /* end if */
      highLevelErr = AEDisposeDesc(&volDesc);
/* Release memory copy of the AE parameter */
      } /* end if !highLevelErr */

   if (fileShareWasOn)
      Start_File_Sharing();

   if (gdragNDropFlag >= 0)    /* Did user drag & drop onto us? */
      guserDone = TRUE;       /* Yes, stop the application */
```

```
   SetCursor(&qd.arrow);       /* Restore the cursor */
   return highLevelErr;       /* Kick back any high-level problems to calling app */

} /* end Core_AE_OpenDoc_Handler() */

/* High-level print event */
   pascal OSErr Core_AE_Print_Handler(AppleEvent *messagein, AppleEvent *reply,
      ➥long refIn)
{
    return errAEEventNotHandled;   /* No printing done here, so no print handler */
} /* end Core_AE_Print_Handler() */

/* High-level quit event  */
   pascal OSErr Core_AE_Quit_Handler(AppleEvent *messagein, AppleEvent *reply,
      ➥long refIn)
{
    guserDone = TRUE;       /* Tell main event loop we want to stop */
    return noErr;
} /* Core_AE_Quit_Handler() */

void Do_High_Level(EventRecord *AERecord)
{
   AEProcessAppleEvent(AERecord);
} /* end Do_High_Level() */

/* Do our checks for system-specific characteristics here. You can use
   the Gestalt Manager for this, but it requires System 7. Here, we're
   using the old SysEnvirons() routine to see if we have System 7. For
   SwitchBank, System 7 alone should have everything we need. */
Boolean Check_System(void)
{
SysEnvRec   machineInfo; /* Record with machine-specific data */
short      sysVersion;         /* System version # */
short      versionRequested;
/* Version of SysEnvirons() to use */

   sysVersion = SYSTEM_7;
   versionRequested = 2;      /* MUST set this value if you want valid results */

    if (SysEnvirons(versionRequested, &machineInfo) == noErr)
      sysVersion = machineInfo.systemVersion;
```

```
   else
      {
      Report_Err_Message(TROUBLE_WITH_SYS_INFO);
      return FALSE;
      } /* end else */

   if (sysVersion < SYSTEM_7)   /* Running System 7.0? */
      {
      Report_Err_Message (SYSTEM_7_REQUIRED);
      return FALSE;         /* No. Sorry, can't run without it  */
      } /* end if */

   return TRUE;
} /* end Check_System() */

/* Handle a command through menu activation. Don't forget to unhighlight the
   selection to indicate the application is done. (Menu is highlighted
   automagically by MenuSelect.) */
Boolean Do_Command (long mResult)
{
unsigned char   accName[255];
short         itemHit;
Boolean         quitApp;
short         refNum;
DialogPtr      theDialog;
short         theItem, theMenu;
GrafPtr         savePort;      /* place to stow current GrafPort when we
                           activate a Desk Accessory (DA) */

   quitApp = FALSE;/* Assume Quit not activated */
   theMenu = HiWord(mResult);   /* Extract the menu selected */
   theItem = LoWord(mResult);   /* Get the item on the menu  */

   switch (theMenu)
      {
      case APPLE_MENU:
         if (theItem == ABOUT_BOX)/* "About..." selected, describe ourself */
            {
            if ((theDialog = GetNewDialog(ABOUT_BOX_ID, NIL, (WindowPtr)
               ➥IN_FRONT)) != NIL)
               {
               ModalDialog(NIL, &itemHit);
               DisposDialog(theDialog);
```

```
            } /* end if != NIL */
         else
            SysBeep(30);
         } /* end if theItem == ABOUT_BOX */
      else              /* It's a DA */
         {
         GetPort(&savePort);    /* Save port (in case the DA doesn't) */
         GetMenuItemText(GetMenuHandle(APPLE_MENU), theItem, accName);
         refNum = OpenDeskAcc(accName);  /* Start it */
         SetPort(savePort);     /* Done, restore the port */
         }
   break; /* end APPLE_MENU case */

   case FILE_MENU:
      switch(theItem)
         {
         case I_QUIT:
            quitApp = TRUE;
         break;
         } /* end switch */
   break; /* end FILE_MENU case */

   case EDIT_MENU:
      SystemEdit(theItem - 1);
   break;

   case SWITCH_MENU:
      Toggle_File_Sharing();
   break;

   default:
     break;
   } /* end switch */

  HiliteMenu(0);       /* Switch off highlighting on the menu just used */
  return quitApp;
} /* end Do_Command() */

/* The main chunk of code that processes events as they occur. Execution remains */
/* in this loop until Do_Command returns TRUE, indicating the user wants to quit. */
/* In most cases, an event should call a subroutine to handle the event, but in */
```

```
/*  this example the actions are so simple most code can be placed in-line. */
void Main_Event_Loop(void)
{
Point      where;

   gdragNDropFlag = 1;
   FlushEvents(everyEvent, 0);   /* Clear out left over events */
   guserDone = FALSE;

   do
      {
      if (WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR))
         {               /* We have an event... */
         switch(gmyEvent.what)/* Field each type of event */
            {
            case mouseDown:   /* In what window, and where?? */
               gwindowCode = FindWindow(gmyEvent.where, &gwhichWindow);
                  switch(gwindowCode)
                  {
                  case inSysWindow:
/* It's a Desk Accessory (DA) */
                     SystemClick(&gmyEvent, gwhichWindow);
                  break;
                  case inDrag:      /* Drag the window */
                  break;
                  case inGrow:
/* Grow the window, if size has changed */
                  break;
                  case inContent:
* Bring window to front if it's not, and that's all */
                  break;
                  case inMenuBar:
/* In a menu, handle the command */
                     guserDone = Do_Command(MenuSelect(gmyEvent.where));
                  break;
                  } /* end switch gwindowCode */
               break; /* end mouseDown */
            case keyDown:
            case autoKey:
/* Command key pressed, pass to MenuKey */
               if((gmyEvent.modifiers & cmdKey) != 0)
                  guserDone = Do_Command(MenuKey((char) (gmyEvent.message
                                 & CHAR_CODE_MASK)));
```

```
                break; /* end key events */
            case updateEvt:   /* Update the window */
                   gwhichWindow = (WindowPtr) gmyEvent.message;
                break;
            case diskEvt:   /* Handle disk insertion event */
               if (HiWord(gmyEvent.message) != noErr)
                  {
                  DILoad();
                  where.h = INIT_X;
                  where.v = INIT_Y;
                  DIBadMount(where, gmyEvent.message);
                  DIUnload();
                  } /* end if != noErr */
            break; /* end disk event */
            case activateEvt:/* Activate event */
                   gwhichWindow = (WindowPtr) gmyEvent.message;
            break;
            case kHighLevelEvent:      /* Handle Apple Event */
               Do_High_Level(&gmyEvent);
            break;
            default:
            break;
            } /* end switch gmyEvent.what */
         } /* end if on next event */
      else               /* Null event */
         ;       /* Do idle or background stuff here */

/* Use this flag to tell Core_AE_OpenDoc_Handler() whether to shut down app when */
/* done (user dragged file onto app) or not (user left app running). We bump this */
/* flag down twice, after which point we stop, because more than 2 events */
/* indicates the app is running */

      if (gdragNDropFlag >= 0)
         gdragNDropFlag--;
      } /* end do */

   while (guserDone == FALSE)
      ;                  /* Loop until told to stop */
} /* end Main_Event_Loop() */
```

```
Boolean Init_Mac(void)
{
Handle      theMenuBar;

/* Lunge after all the memory we can get */
   MaxApplZone();

/* Make sure we've got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(NIL);


   if ((theMenuBar = GetNewMBar(MENU_BAR_ID)) == NIL)
/* Got our menu resources OK? */
      return FALSE;

   SetMenuBar(theMenuBar); /* Add our menus to menu list */
   DisposHandle(theMenuBar);
   AppendResMenu(GetMenuHandle(APPLE_MENU), 'DRVR');
/* Build Apple menu */
   DrawMenuBar();

/* Look for specific features or set up handlers this app needs */
   if (!Check_System())              /* Need System 7 */
      return FALSE;

   if (!Init_AE_Events())
 /* Set up our high-level event handlers */
      return FALSE;
```

```
   if (FindFolder(kOnSystemDisk, kSystemFolderType, kDontCreateFolder,
                        &gsysVRefNum, &gSysDirID) != noErr)
      {
      Report_Err_Message (CANT_FIND_STARTUP_VOL);
      return FALSE;
      } /* end if */

   InitCursor();    /* Tell user app is ready */
   return TRUE;

} /* end Init_Mac() */

void main(void)
{
   if (Init_Mac())
      Main_Event_Loop();
   else
      SysBeep(30);
} /* end main */

/* Glue to call the ServerDispatch trap */
#if USES68KINLINES
#pragma parameter __D0 mySyncServerDispatch(__A0)
#endif
pascal OSErr mySyncServerDispatch(SCParamBlockPtr PBPtr)
 FOURWORDINLINE(0x7000, 0xA094, 0x3028, 0x0010);

/*    = {                                                          */
/*      0x7000,/* MOVEQ      #$00,  // Input must be 0             */
/*      0xA094,/* _ServerDispatch  // Hop to the trap             */
/*      0x3028,0x0010    /* MOVE.W   ioResult(A0),D0  // Move result to D0 because */
/*      };                       //    File Sharing doesn't.     */

#ifdef powerc
/* Call the 68K code from the PowerPC through the Mixed Mode Manager */
static pascal OSErr mySyncServerDispatch(SCParamBlockPtr PBPtr)
{
ProcInfoType   myProcInfo;
OSErr         result;
/* Need an RTS at the end to return ... */
static short code[] = {0x7000, 0xA094, 0x3028, 0x0010, 0x4E75};
```

```
   /* Build the procinfo (note use of register based calls) */
   myProcInfo = kRegisterBased
            | RESULT_SIZE(SIZE_CODE(sizeof(OSErr)))
            | REGISTER_RESULT_LOCATION(kRegisterD0)
            |
REGISTER_ROUTINE_PARAMETER(1,kRegisterA0,SIZE_CODE(sizeof(SCParamBlockPtr)));

   result = CallUniversalProc((UniversalProcPtr) code, myProcInfo, (PBPtr));
   return result;
} /* mySyncServerDispatch() */
#endif

/* Send a shut down immediately message to the File Sharing Server */
void Stop_File_Sharing(void)
{
DisconnectParam   serverBlock;
SCParamBlockPtr   serverBlockPtr;

   serverBlockPtr = (SCParamBlockPtr) &serverBlock;
/* Point to our message block */
   serverBlock.scCode = SHUT_DOWN;
/* Server command to shut down */
   serverBlock.scNumMinutes = 0; /* Do it immediately */
   serverBlock.scFlags = SEND_MESSAGE;
   serverBlock.scMessagePtr = NIL;

   if (mySyncServerDispatch(serverBlockPtr) == noErr)
      {
      WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
/* Let the OS get at the event */
      WaitNextEvent(everyEvent, &gmyEvent, LONG_NAP, NO_CURSOR);
      } /* end if */
   else
      Report_Err_Message(PROBLEM_STOPPING_FS);

} /* end Stop_File_Sharing() */
```

SwitchBank.r

```
#include "SysTypes.r"
#include "Types.r"
```

```
#define AllItems   0b11111111111111111111111111111111
/* 31 flags */
#define NoItems      0b00000000000000000000000000000000
#define MenuItem1   0b00000000000000000000000000000001
#define MenuItem2   0b00000000000000000000000000000010
#define MenuItem3   0b00000000000000000000000000000100
#define MenuItem4   0b00000000000000000000000000001000

#define MENU_BAR_ID         128
/* Menu bar resource for our menus */
#define APPLE_MENU         128 /* Menu ID for Apple menu */
#define FILE_MENU         129 /* Menu ID for File menu */
#define EDIT_MENU         130
/* Menu ID for Edit menu */
#define SWITCH_MENU         131
/* Menu ID for File Share control */

#define ABOUT_BOX_ID      128
/* Resource IDs for our windows & dialogs */
#define ERROR_BOX_ID      130
#define ERROR_MESS_ID      131

#define APPL_FREF         128
/* Resource IDs for file refs & icons */
#define DISK_FREF         129
#define SWITCH_ICON         128

/* Version info for the Finder's Get Info box
resource 'vers' (1, purgeable)
{
   0x01,
   0x10,
   beta,
   0x00,
   verUs,
   "1.1ß",
   "1.1ß, by Tom Thompson"
};

/* Menu resources */
resource 'MBAR' (MENU_BAR_ID, preload)
{
```

```
   { APPLE_MENU, FILE_MENU, EDIT_MENU, SWITCH_MENU };
};

resource 'MENU' (APPLE_MENU, preload)
{
   APPLE_MENU, textMenuProc,
   AllItems & ~MenuItem2,
/* Disable separator line, enable About Box and DAs */
   enabled, apple,
   {
      "About SwitchBank 1.1…",   noicon, nokey, nomark, plain;
      "-",                  noicon, nokey, nomark, plain
   }
};

resource 'MENU' (FILE_MENU, preload)
{
   FILE_MENU, textMenuProc,
   AllItems,
   enabled, "File",
   {
      "Quit",                  noicon, "Q", nomark, plain
   }
};

resource 'MENU' (EDIT_MENU, preload)
{
   EDIT_MENU, textMenuProc,
   AllItems & ~MenuItem2,      /* Disable separator line */
   enabled, "Edit",
   {
      "Undo",                  noicon, "Z", nomark, plain;
      "-",                  noicon, nokey, nomark, plain;
      "Cut",                  noicon, "X", nomark, plain;
      "Copy",                  noicon, "C", nomark, plain;
      "Paste",               noicon, "V", nomark, plain
   }
};

resource 'MENU' (SWITCH_MENU, preload)
{
   SWITCH_MENU, textMenuProc,
   AllItems,
```

```
   enabled, "Controls",
   {
      "Toggle File Sharing",      noicon, "T", nomark, plain
   }
};

/* Our error messages */
resource 'STR#'(128, purgeable)
{
   {
   /* [1] */   "A problem occured stopping File Sharing.";
   /* [2] */   "A problem occured starting File Sharing.";
   /* [3] */   "A problem occured while ejecting the volume.";
   /* [4] */   "You can't eject the startup volume.";
   /* [5] */   "Couldn't find the startup volume.";
   /* [6] */   "Couldn't get valid system information.";
   /* [7] */   "Couldn't locate the File Sharing Extension file.";
   /* [8] */   "A problem occurred while loading the Apple Event handlers.";
   /* [9] */   "Sorry, SwitchBank requires System 7 or later to run.";
   }
};

/* This ALRT and DITL are used as an About Box */
resource 'DLOG' (ABOUT_BOX_ID, purgeable)
{
   {31, 6, 224, 265},
   altDBoxProc,
   visible,
   noGoAway,
   0x0,         /* No refCon */
   ABOUT_BOX_ID,
   ""            /* No window title */
};

resource 'DITL' (ABOUT_BOX_ID, purgeable)
{
   {
      /* Item 1 */
      {154, 80, 175, 180},   Button { enabled, "OK" },
      /* Item 2 */
      {4, 68, 38, 193},      StaticText { disabled, " SwitchBank 1.1\nby Tom
      ➥Thompson" },
      /* Item 3 */
```

```
        {86, 11, 102, 250},      StaticText { disabled, " Copyright © 1994 Tom
           ➥Thompson." },
        /* Item 4 */
        {44, 114, 76, 146},      Icon { disabled, 128 },
        /* Item 5 */
        {107, 43, 133, 217},   StaticText { disabled, "Written in Metrowerks C " }
    }
};

/* The ALRT and DITL for the basic error screen */
resource 'ALRT' (ERROR_BOX_ID, purgeable)
{
    {40, 40, 127, 273},
    ERROR_BOX_ID,
    {
        OK, visible, silent,
        OK, visible, silent,
        OK, visible, silent,
        OK, visible, silent
    }
};

resource 'DITL' (ERROR_BOX_ID, purgeable)
{
    {
        { 52, 162, 72, 220 },      Button { enabled, "OK" },
        { 54, 17, 70, 151 },      StaticText { disabled, "I/O error, ID = ^0" }
    }
};

/* Alert and DITL for error message screen */
resource 'ALRT' (ERROR_MESS_ID, purgeable)
{
    { 40, 40, 147, 280 },
    ERROR_MESS_ID,
    {
        OK, visible, silent,
        OK, visible, silent,
        OK, visible, silent,
        OK, visible, silent
    }
};
```

```
resource 'DITL' (ERROR_MESS_ID, purgeable)
{
   {
      { 73, 168, 93, 226 },      Button { enabled, "OK" },
      { 53, 14, 97, 157 },      StaticText { disabled, "^0" }
   }
};

/* File reference resources */
resource 'FREF' (DISK_FREF)
{
   'disk',
   1,
   ""
};


resource 'FREF' (APPL_FREF)
{
   'APPL',
   0,
   ""
};

/* Bundle resource */
resource 'BNDL' (128)
{
   'SWCH', 0,
   {
   'ICN#', { 0, SWITCH_ICON },      /* Only 1 icon */
   'FREF', { 0, APPL_FREF, 1, DISK_FREF }
 /* Two types of files */
   }
};

/* Signature resource - all 'STR ' resources must be declared before this! */
type 'SWCH' as 'STR ';

resource 'SWCH' (0) {
   "SwitchBank 1.1ß"
};
```

```
/* Our icon data */
data 'ICON' (SWITCH_ICON)
{
   $"7FFF FFFE 4000 0002 5C00 003A 55F8 1FAA"
   $"5D08 10BA 4108 1082 4108 1082 4108 1082"
   $"41B8 1D82 4110 0882 4110 0882 4110 0882"
   $"471C 38E2 4514 28A2 4514 28A2 4514 28A2"
   $"471C 38E2 4110 0882 411F F882 4110 0882"
   $"4110 0882 4110 0882 41FF FF82 4004 2002"
   $"4004 2002 4004 2002 4004 2002 5C04 203A"
   $"5404 202A 5C07 E03A 4000 0002 7FFF FFFE"
};

data 'ICN#' (SWITCH_ICON)
{
   $"7FFF FFFE 4000 0002 5C00 003A 55F8 1FAA"
   $"5D08 10BA 4108 1082 4108 1082 4108 1082"
   $"41B8 1D82 4110 0882 4110 0882 4110 0882"
   $"471C 38E2 4514 28A2 4514 28A2 4514 28A2"
   $"471C 38E2 4110 0882 411F F882 4110 0882"
   $"4110 0882 4110 0882 41FF FF82 4004 2002"
   $"4004 2002 4004 2002 4004 2002 5C04 203A"
   $"5404 202A 5C07 E03A 4000 0002 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
   $"7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE"
};

/* SwitchBank's color icon in icl8 format */
data 'icl8' (SWITCH_ICON)
{
   $"00FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FF00"
   $"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A"
   $"2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
   $"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2A2A 2A2A"
   $"2A2A 2A2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
   $"00FF 2AFF 2AFF 2AFF FFFF FFFF FF2A 2A2A"
```

```
$"2A2A 2AFF FFFF FFFF FF2A FF2A FF2A FF00"
$"00FF 2AFF FFFF 2AFF F52A F52A FF2A 2A2A"
$"2A2A 2AFF F52A F52A FF2A FFFF FF2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A2A 2A2A FF2A 2A2A"
$"2A2A 2AFF 2A2A 2A2A FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 5454 5454 FF2A 2A2A"
$"2A2A 2AFF 5454 5454 FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 7F7F 7F7F FF2A 2A2A"
$"2A2A 2AFF 7F7F 7F7F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF FF7F FFFF FF2A 2A2A"
$"2A2A 2AFF FFFF 7FFF FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 7F7F 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF7F 7F7F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 5454 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF54 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2AFF FFFF 2A54 7FFF FFFF 2A2A"
$"2A2A FFFF FF2A 547F FFFF FF2A 2A2A FF00"
$"00FF 2A2A 2AFF F5FF 2A54 7FFF F5FF 2A2A"
$"2A2A FFF5 FF2A 547F FFF5 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF 54FF 2A54 7FFF 54FF 2A2A"
$"2A2A FF54 FF2A 547F FF54 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF 54FF 2A54 7FFF 54FF 2A2A"
$"2A2A FF54 FF2A 547F FF54 FF2A 2A2A FF00"
$"00FF 2A2A 2AFF FFFF 2A54 7FFF FFFF 2A2A"
$"2A2A FFFF FF2A 547F FFFF FF2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 2A2A 2A2A"
$"2A2A 2A2A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF FFFF FFFF"
$"FFFF FFFF FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF F52A F52A"
$"F52A F52A FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 5454 5454"
$"5454 5454 FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF 2A54 7FFF 7F7F 7F7F"
$"7F7F 7F7F FF2A 547F FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2AFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FF2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 54F5"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
$"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 542A"
$"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
```

```
    $"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 54F5"
    $"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
    $"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2AFF 542A"
    $"2A7F FF2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
    $"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2AFF 54F5"
    $"2A7F FF2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
    $"00FF 2AFF 2AFF 2A2A 2A2A 2A2A 2AFF 542A"
    $"2A7F FF2A 2A2A 2A2A 2A2A FF2A FF2A FF00"
    $"00FF 2AFF FFFF 2A2A 2A2A 2A2A 2AFF FFFF"
    $"FFFF FF2A 2A2A 2A2A 2A2A FFFF FF2A FF00"
    $"00FF 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A FF00"
    $"00FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FF00"
};

/* SwitchBank's color icon, in cicn format */
data 'cicn' (SWITCH_ICON)
{
    $"0000 0000 8010 0000 0000 0020 0020 0000"
    $"0000 0000 0000 0048 0000 0048 0000 0000"
    $"0004 0001 0004 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0004 0000 0000 0020 0020"
    $"0000 0000 0004 0000 0000 0020 0020 0000"
    $"0000 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 7FFF FFFE 7FFF FFFE 7FFF"
    $"FFFE 7FFF FFFE 4000 0002 5C00 003A 55F8"
    $"1FAA 5D08 10BA 4108 1082 4108 1082 4108"
    $"1082 41B8 1D82 4110 0882 4110 0882 4110"
    $"0882 471C 38E2 4514 28A2 4514 28A2 4514"
    $"28A2 471C 38E2 4110 0882 411F F882 4110"
    $"0882 4110 0882 4110 0882 41FF FF82 4004"
    $"2002 4004 2002 4004 2002 4004 2002 5C04"
    $"203A 5404 202A 5C07 E03A 4000 0002 7FFF"
    $"FFFE 0000 0000 0000 0005 0000 FFFF FFFF"
    $"FFFF 0001 CCCC CCCC FFFF 0002 9999 9999"
    $"FFFF 0003 6666 6666 CCCC 0004 EEEE EEEE"
```

```
   $"EEEE 000F 0000 0000 0000 0FFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFF0 0F11 1111 1111"
   $"1111 1111 1111 1111 11F0 0F1F FF11 1111"
   $"1111 1111 1111 11FF F1F0 0F1F 1F1F FFFF"
   $"F111 111F FFFF F1F1 F1F0 0F1F FF1F 4141"
   $"F111 111F 4141 F1FF F1F0 0F11 111F 1111"
   $"F111 111F 1111 F111 11F0 0F11 111F 2222"
   $"F111 111F 2222 F111 11F0 0F11 111F 3333"
   $"F111 111F 3333 F111 11F0 0F11 111F F3FF"
   $"F111 111F FF3F F111 11F0 0F11 111F 333F"
   $"1111 1111 F333 F111 11F0 0F11 111F 223F"
   $"1111 1111 F223 F111 11F0 0F11 111F 123F"
   $"1111 1111 F123 F111 11F0 0F11 1FFF 123F"
   $"FF11 11FF F123 FFF1 11F0 0F11 1F4F 123F"
   $"4F11 11F4 F123 F4F1 11F0 0F11 1F2F 123F"
   $"2F11 11F2 F123 F2F1 11F0 0F11 1F2F 123F"
   $"2F11 11F2 F123 F2F1 11F0 0F11 1FFF 123F"
   $"FF11 11FF F123 FFF1 11F0 0F11 111F 123F"
   $"1111 1111 F123 F111 11F0 0F11 111F 123F"
   $"FFFF FFFF F123 F111 11F0 0F11 111F 123F"
   $"4141 4141 F123 F111 11F0 0F11 111F 123F"
   $"2222 2222 F123 F111 11F0 0F11 111F 123F"
   $"3333 3333 F123 F111 11F0 0F11 111F FFFF"
   $"FFFF FFFF FFFF F111 11F0 0F11 1111 1111"
   $"1F24 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F21 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F24 13F1 1111 1111 11F0 0F11 1111 1111"
   $"1F21 13F1 1111 1111 11F0 0F1F FF11 1111"
   $"1F24 13F1 1111 11FF F1F0 0F1F 1F11 1111"
   $"1F21 13F1 1111 11F1 F1F0 0F1F FF11 1111"
   $"1FFF FFF1 1111 11FF F1F0 0F11 1111 1111"
   $"1111 1111 1111 1111 11F0 0FFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFF0"
};

/* The system's color caution alert icon */
data 'cicn' (2)
{
   $"0000 0000 8010 0000 0000 0020 0020 0000"
   $"0000 0000 0000 0048 0000 0048 0000 0000"
   $"0004 0001 0004 0000 0000 0000 0000 0000"
   $"0000 0000 0000 0004 0000 0000 0020 0020"
   $"0000 0000 0004 0000 0000 0020 0020 0000"
```

```
$"0000 0001 8000 0003 C000 0007 E000 0007"
$"E000 000F F000 000F F000 001F F800 001F"
$"F800 003F FC00 003F FC00 007F FE00 007F"
$"FE00 00FF FF00 00FF FF00 01FF FF80 01FF"
$"FF80 03FF FFC0 03FF FFC0 07FF FFE0 07FF"
$"FFE0 0FFF FFF0 0FFF FFF0 1FFF FFF8 1FFF"
$"FFF8 3FFF FFFC 3FFF FFFC 7FFF FFFE 7FFF"
$"FFFE FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF 0001 8000 0003 C000 0003 C000 0006"
$"6000 0006 6000 000C 3000 000C 3000 0018"
$"1800 0019 9800 0033 CC00 0033 CC00 0063"
$"C600 0063 C600 00C3 C300 00C3 C300 0183"
$"C180 0183 C180 0303 C0C0 0303 C0C0 0603"
$"C060 0601 8060 0C01 8030 0C00 0030 1800"
$"0018 1801 8018 3003 C00C 3003 C00C 6001"
$"8006 6000 0006 C000 0003 FFFF FFFF 7FFF"
$"FFFE 0000 0000 0000 0006 0000 FFFF FFFF"
$"FFFF 0001 FFFF CCCC 3333 0002 CCCC 9999"
$"0000 0003 9999 6666 0000 0004 3333 3333"
$"3333 0005 BBBB BBBB BBBB 000F 0000 0000"
$"0000 0000 0000 0000 000F F000 0000 0000"
$"0000 0000 0000 0000 004F F400 0000 0000"
$"0000 0000 0000 0000 05FF FF50 0000 0000"
$"0000 0000 0000 0000 04F3 3F40 0000 0000"
$"0000 0000 0000 0000 5FF1 1FF5 0000 0000"
$"0000 0000 0000 0000 4F31 13F4 0000 0000"
$"0000 0000 0000 0005 FF11 11FF 5000 0000"
$"0000 0000 0000 0004 F311 113F 4000 0000"
$"0000 0000 0000 005F F12F F21F F500 0000"
$"0000 0000 0000 004F 314F F413 F400 0000"
$"0000 0000 0000 05FF 11FF FF11 FF50 0000"
$"0000 0000 0000 04F3 11FF FF11 3F40 0000"
$"0000 0000 0000 5FF1 11FF FF11 1FF5 0000"
$"0000 0000 0000 4F31 11FF FF11 13F4 0000"
$"0000 0000 0005 FF11 11FF FF11 11FF 5000"
$"0000 0000 0004 F311 11FF FF11 113F 4000"
$"0000 0000 005F F111 11FF FF11 111F F500"
$"0000 0000 004F 3111 11FF FF11 1113 F400"
$"0000 0000 05FF 1111 11FF FF11 1111 FF50"
$"0000 0000 04F3 1111 114F F411 1111 3F40"
$"0000 0000 5FF1 1111 112F F211 1111 1FF5"
$"0000 0000 4F31 1111 111F F111 1111 13F4"
$"0000 0005 FF11 1111 1112 2111 1111 11FF"
```

```
    $"5000 0004 F311 1111 1111 1111 1111 113F"
    $"4000 005F F111 1111 112F F211 1111 111F"
    $"F500 004F 3111 1111 11FF FF11 1111 1113"
    $"F400 05FF 1111 1111 11FF FF11 1111 1111"
    $"FF50 04F3 1111 1111 112F F211 1111 1111"
    $"3F40 5FF1 1111 1111 1111 1111 1111 1111"
    $"1FF5 FF31 1111 1111 1111 1111 1111 1111"
    $"13FF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF 5FFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFF5"
};
```

FlipDepth.c

```
/*
   Portions © 1994 Rock Ridge Enterprises. All Rights Reserved.
*/


/*
   This #define is for testing only. Without it, only the
   68k version of our patch is called.
*/
#undef DO_PPC_CODE_ONLY
//#define DO_PPC_CODE_ONLY

#define USESROUTINEDESCRIPTORS GENERATINGCFM
#pragma once on

#include <Memory.h>
#include <Gestalt.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Files.h>
#include <Devices.h>
#include <Resources.h>
#include <Errors.h>
#include <OSUtils.h>
#include <Traps.h>
#include <LowMem.h>
#include <Events.h>
```

```
#include <Palettes.h>
#include <MixedMode.h>
#include <ConditionalMacros.h>
#include <CodeFragments.h>

#ifndef powerc
   #include <A4Stuff.h>
   #include <SetUpA4.h>
#endif

#define FALSE            false
#define TRUE            true
#define NIL               0L

/*
   Some low memory globals. We'd rather not use these, but they're
   necessary because we'll be operating in a trap that doesn't move memory.
*/

#define KeyMapLM         0x174
#define lowMemKeyStroke      (*(KeyMap *) KeyMapLM)[0]
#define lowMemKeyModifiers   (*(KeyMap *) KeyMapLM)[1]

/* Some constants that define the bits we'll see in KeyMap */
#define SHIFT_KEY         1L
#define CAPS_LOCK         2L
#define OPTION_KEY         4L
#define CONTROL_KEY         8L
#define COMMAND_KEY         0x8000L

#define KEY_COMBO         SHIFT_KEY + COMMAND_KEY
#define T_KEYCODE         0x0200L
#define BLACK_WHITE         128
/* First video mode ID in sResource list */


#define kOldSystemErr      10000

/*==========================
   We take the PowerPC code from the data fork
   and put it into a resource using a utility
   like Resorcerer.
==========================*/
```

```
#define kPPCRezType          'PPC '
#define kPPCRezID          300

/*============================
   The 68k code goes in a normal INIT resource.
   Be sure this is set to "system heap/locked".
============================*/
#define kInitRezType       'INIT'
#define kInitRezID          128

#define kMinSystemVersion    (0x0605)

/*============================
   This is the name of the ppc fragment - for debugging only.
============================*/
#define kInitName       "\pEricsInit"

/*============================
   to save some screen space, we'll use "UPP" instead of "UniversalProcPtr"
============================*/
typedef UniversalProcPtr   UPP;

/*============================
   PostEvent Information
============================ */
enum
{
   kPostEventInfo = kRegisterBased
            | RESULT_SIZE(SIZE_CODE(sizeof(OSErr)))
            | REGISTER_RESULT_LOCATION(kRegisterD0)
            | REGISTER_ROUTINE_PARAMETER(1, kRegisterA0, SIZE_CODE(sizeof(short)))
            | REGISTER_ROUTINE_PARAMETER(2, kRegisterD0, SIZE_CODE(sizeof(long)))
};


typedef pascal OSErr ( *PostEventFuncPtr ) ( short eventNum, long eventMsg );
#define kPostEventFuncName "\pMyPostEventPPC"

/* Note separate functions */
short MyPostEvent68k( short eventNum, long eventMsg );
//OSErr MyPostEventPPC( unsigned short trapNum, short eventNum, long eventMsg );
OSErr MyPostEventPPC( short eventNum, long eventMsg );
```

```
/*==========================
   GetMouse Information
========================== */
enum
{
   kGetMouseInfo = kPascalStackBased
            ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Point)))
};

typedef pascal void ( *GetMouseFuncPtr ) ( Point *mouseLoc );
#define kGetMouseFuncName "\pMyGetMouse"
void MyGetMouse ( Point *mouseLoc );   /* Only one function required */

 /*==========================
   An original trap is called differently from PowerPC
   code than from 68k code because CallUniversalProc() and
   CallOSTrapUniversalProc isn't implemented for 68k code.
==========================*/
#ifdef powerc
   #define CallPostEvent(eventNum, eventMsg)
      ➥CallOSTrapUniversalProc( gGlobalsPtr->gOrigPostEvent,
      ➥kPostEventInfo, eventNum, eventMsg )
   #define CallGetMouse(mouseLoc)   CallUniversalProc( gGlobalsPtr->gOrigGetMouse,
      ➥kGetMouseInfo, mouseLoc )
#else
   #define CallGetMouse(mouseLoc)
(*(GetMouseFuncPtr)gGlobalsPtr->gOrigGetMouse)( mouseLoc );
#endif


/*==========================
   ShowInitIcon() defintions
========================== */
enum
{
   kShowInitIconInfo = kPascalStackBased
               ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
               ¦ STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(Boolean)))
};

// Function to display startup icon

typedef   pascal void (*ShowInitProcPtr) (short iconFamilyID, Boolean advance);

#if powerc
```

```
typedef UPP ShowInitIconProcUPP;

#define CallShowInitIconProc(userRoutine, iconFamilyID, advance)      \
      CallUniversalProc( (UPP)(userRoutine), kShowInitIconInfo, (iconFamilyID),
       ➥(advance) )
#else
typedef ShowInitProcPtr ShowInitIconProcUPP;

#define  CallShowInitIconProc(userRoutine, iconFamilyID, advance)      \
       (*(userRoutine))((iconFamilyID), (advance))
#endif

// Here we supply in-line definitions for NewRoutineDescriptor() and
     ➥NewFatRoutineDescriptor()
// for a finicky linker

 #ifdef NewRoutineDescriptor
   #undef NewRoutineDescriptor
#endif
#ifdef NewFatRoutineDescriptor
   #undef NewFatRoutineDescriptor
#endif

extern pascal UPP NewFatRoutineDescriptor(ProcPtr theM68kProc, ProcPtr
    ➥thePowerPCProc, ProcInfoType theProcInfo)
 TWOWORDINLINE(0x7002, 0xAA59);

extern pascal UPP NewRoutineDescriptor(ProcPtr theProc, ProcInfoType theProcInfo,
     ➥ISAType theISA)
TWOWORDINLINE (0x7000, 0xAA59);

/* Custom function to place our patch code in the system heap */
Handle Get1ResourceSys( OSType rezType, short rezID );

/* Functions that change screen depth. Works one both platforms. */
void Change_Depth(long newDepth);
long Fetch_Depth(void);


/*==========================
   This structure is shared between the power pc
   version of the code and the 68k version.
```

```
   Both the PowerPC code and the 68k code have a single
   global variable, "gGlobalsPtr". They point to the
   same area of memory.
============================*/

#ifdef powerc
   #pragma options align=mac68k
#endif

/*
   Note: do not move these fields around!
   The assembly code in PostEvent68kStub()
   depends on their locations. It must be
   compiled with the 68K packing conventions
*/

typedef struct
{
UPP            gOrigPostEvent;
/* Address of original PostEvent trap */
UPP            gOrigGetMouse;
/* Address of original GetMouse trap */
SysEnvRec      gSystemInfo;
Boolean         gRequestFlag;
/* Flag that signals screen depth change */
GDHandle      gOurGDevice;
/* The GDevice of the screen we're working with */
short           gDevRefNum;
/* Driver ref number for video board's slot */
long         gOldScreenDepth;
} MyInitGlobals;

#ifdef powerc
   #pragma options align=reset
#endif

/*==========================
   Global Variables

    -- each side of the code maintains its own pointer to the
       same block of memory.

    -- we reference the globals ptr by name, so these two must be
       changed together.
```

```
============================*/
MyInitGlobals            *gGlobalsPtr;
#define kGlobalsSymName      "\pgGlobalsPtr"

/*
   @@@@@@@@@@@@@@@  68000 Exclusive Code   @@@@@@@@@@@@@@@
*/
#ifndef powerc

/*============================
   Prototypes for 68k code
============================*/
OSErr       DoInitForOldMacs( void );
OSErr       DoInitForPPCMacs( void );
OSErr       CreateFatDescriptorSys( void *mac68Code, void *ppcCode,
                          ProcInfoType procInfo, UPP *result );
OSErr       PatchTrapsForPPCMac( ConnectionID connID );

void      PostEvent68kStub( void );
pascal void GetMouse68kStub ( Point *mouseLoc );

/*============================
   This is *always* the INIT's entry point. This is
   the only routine called by system software at startup.

   This requires that the INIT resource be set to
   System Heap/Locked.
============================*/
void main( void )
{
long            oldA4;
Handle            initH = nil;
/* Handle to our own INIT resource */
OSErr            err = noErr;
long            ginfo;
ShowInitIconProcUPP   showCode;
Handle            showResource;

   /*******************************
      global variable support
      Place proper value for A4 into hole in INIT resource.
   *******************************/
```

```
   oldA4 = SetCurrentA4();
/* Get the proper value of A4 into A4 */
   RememberA4();
/* save into self-modifying code */

   /*******************************
      Allocate our global variables
   *******************************/
   gGlobalsPtr = (MyInitGlobals*) NewPtrSysClear( sizeof(MyInitGlobals) );
   if ( !gGlobalsPtr )
      {
      err = memFullErr;
      goto DONE;
      }

   /*******************************
      Get some basic system information
   *******************************/
   err = SysEnvirons( 1, &gGlobalsPtr->gSystemInfo );
   if ( err )
      goto DONE;

   /*******************************
      Check the system version
   *******************************/
    if ( gGlobalsPtr->gSystemInfo.systemVersion < kMinSystemVersion )
       {
       err = kOldSystemErr;
       goto DONE;
       }

   /*******************************
      Get a handle to our own INIT resource
   *******************************/
   initH = Get1Resource( kInitRezType, kInitRezID );
   if ( !initH )
      {
      err = resNotFound;
      goto DONE;
      }

   /*******************************
      See if we're running on a PowerPC
   *******************************/
```

```
    err = Gestalt( gestaltSysArchitecture, &ginfo );

    /*******************************
       Patch all the traps and get everything ready.
    *******************************/
    if ( err || (ginfo == gestalt68k) )
       err = DoInitForOldMacs();
    else
       err = DoInitForPPCMacs();

    DONE:
    showResource = Get1Resource('sdes', 128 );   // Get ShowInitIcon() code
    if (showResource != NIL)
       showCode = (ShowInitIconProcUPP) (*showResource);


// Get pointer to resource header

// Don't need to lock down

 //   the resource because its

 //   attribute flags are marked

  //   as Locked and SysHeap

    if ( err )   // Something went wrong, clean up and display failure icon
       {
       if ( gGlobalsPtr )
          DisposPtr( (Ptr)gGlobalsPtr );

       if (showResource != NIL)
          CallShowInitIconProc(showCode, (kInitRezID + 1), TRUE );
// Display bad load icon
       }
    else
// No initialization problems, do final setup and display success icon
       {
       gGlobalsPtr->gRequestFlag = FALSE;
```

```
// Clear request flag
      gGlobalsPtr->gOldScreenDepth = Fetch_Depth();
// Get screen depth for later

// Make sure the INIT code stays in memory when the Extension file closes. We
// do nothing
// for showResource because we want it purged.

      DetachResource( initH );

      if (showResource != NIL)
         CallShowInitIconProc(showCode, kInitRezID, TRUE);
// Display success icon
      } /* end else */

   RestoreA4( oldA4 );      /* restore previous value of A4 */
} /* end main() */

/*============================
   DoInitForOldMacs

   Initialization code for non-powerpc Macs.
============================*/
OSErr DoInitForOldMacs( void )
{
   /* patch the traps */

   gGlobalsPtr->gOrigPostEvent = GetOSTrapAddress( _PostEvent );
   SetOSTrapAddress( (UPP)PostEvent68kStub, _PostEvent );
   gGlobalsPtr->gOrigGetMouse = GetToolTrapAddress( _GetMouse );
   SetToolTrapAddress( (UPP)GetMouse68kStub, _GetMouse );

   return noErr;
} /* end DoInitForOldMacs() */

/*============================
   DoInitForPPCMacs

   Initialization code for powerpc Macs.
============================*/
OSErr DoInitForPPCMacs( void )
{
```

```
OSErr         err = noErr;
Handle         ppcCodeH = nil;
SymClass      theSymClass;
Ptr            theSymAddr;
ConnectionID   connID = kNoConnectionID;
Str255         errName;
Ptr            mainAddr;

   /*******************************
      load the powerpc version of the code into
      memory. since some of our trap patches may be
      called at interrupt time, don't use disk-based
      versions of the code.
   *******************************/
   ppcCodeH = Get1ResourceSys( kPPCRezType, kPPCRezID );
   if ( !ppcCodeH )
      return resNotFound;
   HLock( ppcCodeH );

   /*******************************
      open a connection with the code fragment we just loaded
   *******************************/
   err = GetMemFragment(   *ppcCodeH, GetHandleSize(ppcCodeH), kInitName,
                     kLoadNewCopy, &connID, &mainAddr, errName );
   if ( err )
      {
      connID = kNoConnectionID;
      goto DONE;
      }

   /*******************************
      find the global variable ptr that the powerpc
      code uses.
   *******************************/
   err = FindSymbol( connID, kGlobalsSymName, &theSymAddr, &theSymClass );
   if ( err )
      goto DONE;

   /*******************************
      Modify the powerpc global variable pointer to point
      to the area of memory we've already allocated.
   *******************************/
   *(MyInitGlobals **)theSymAddr = gGlobalsPtr;
```

```
   err = PatchTrapsForPPCMac( connID );

   /*******************************
      Cleanup
   *******************************/
   DONE:
   if ( err )
      {
/* Close the code frag mgr connection if we got an error... */
      if ( connID != kNoConnectionID )
         CloseConnection( &connID );

 /* ...and release the memory we allocated */
      if ( ppcCodeH )
         ReleaseResource( ppcCodeH );
      } /* end if */
   else
      {
/* No error -> keep the ppc code around when file closes */
      DetachResource( ppcCodeH );
      } /* end else */

   return err;
} /* end DoInitForPPCMacs() */

/*==========================
   PatchTrapsForPPCMac
==========================*/
OSErr PatchTrapsForPPCMac( ConnectionID connID )
{
Ptr               symAddr;
SymClass          symType;
OSErr             err = noErr;
UniversalProcPtr   upp = nil;

//      Fat Patch _PostEvent

   err = FindSymbol( connID, kPostEventFuncName, &symAddr, &symType );
   if ( err )
      return err;

   err = CreateFatDescriptorSys( PostEvent68kStub, symAddr, kPostEventInfo, &upp );
```

```
   if ( err )
      return memFullErr;

   gGlobalsPtr->gOrigPostEvent = GetOSTrapAddress( _PostEvent );
   SetOSTrapAddress( (UPP)PostEvent68kStub, _PostEvent );

//      Fat Patch _GetMouse

   err = FindSymbol( connID, kGetMouseFuncName, &symAddr, &symType );
   if ( err )
      return err;

   err = CreateFatDescriptorSys( GetMouse68kStub, symAddr, kGetMouseInfo, &upp );
   if ( err )
      return memFullErr;

   gGlobalsPtr->gOrigGetMouse = GetToolTrapAddress( _GetMouse );
   SetToolTrapAddress( (UPP)GetMouse68kStub, _GetMouse );

   return noErr;
} /* end PatchTrapsForPPCMac() */


/*============================
   CreateFatDescriptorSys

   Creates a fat routine descriptor in the system heap.
============================*/
  OSErr CreateFatDescriptorSys( void *mac68Code, void *ppcCode,
  ➥ ProcInfoType procInfo, UPP *result )
{
   THz      oldZone;
   OSErr   err = noErr;

   oldZone = GetZone();      /* Save current zone */
   SetZone( SystemZone() );   /* Get us in the system heap */

#ifndef DO_PPC_CODE_ONLY
   *result = NewFatRoutineDescriptor( mac68Code, ppcCode, procInfo );
#else
   *result = NewRoutineDescriptor( ppcCode, procInfo, kPowerPCISA );
/* debugging only */
#endif
```

```
    SetZone( oldZone );

    return( *result ? noErr : memFullErr );
} /* end CreateFatDescriptorSys() */


/*==========================
    PostEvent68kStub

    This is the 68k version of PostEvent. Because it's a
    register-based trap, we have to use assembly code
    to see what was passed to it. Because the routine
    can't move memory (it might get called during an
    interrupt), we also have to call a custom 68K
    function that doesn't disturb the machine environment.
==========================*/

asm void PostEvent68kStub( void )
{
        // reserve space on stack for "real" PostEvent address
    sub.l       #4, SP

        // save registers (not A0 & D0, though)
    movem.l       A1-A5/D1-D7, -(SP)

        // push A0 & D0 on stack for call to MyPostEvent68k below
        // we must do this before SetUpA4 since it modifies registers
    move.l      D0, -(SP)          // push event message
    move.w      A0, -(SP)          // push event code

    jsr         SetUpA4             // give us global access

        // put address of "real" postevent in place reserved on stack
        // note that it is the first field in the gGlobals structure
    move.l      gGlobalsPtr, A0
    move.l      (A0), 54(SP)

        // call MyPostEvent68k
        // parameters are on the stack already
        // D0.w returns with the new event code
    jsr         MyPostEvent68k
```

```
    move.w      D0, A0            // A0.w = event code
    add.l      #2, SP            // clear old event code from stack
    move.l      (SP)+, D0         // restore event message from stack

       // restore registers
    movem.l     (SP)+, A1-A5/D1-D7

       // jump directly to original PostEvent code
       // the address was placed on the stack in the above code
    rts
} /* end PostEvent68kStub() */

pascal void GetMouse68kStub( Point *mouseLoc )
{
long      oldA4;

    oldA4 = SetUpA4();
    MyGetMouse ( mouseLoc );
    RestoreA4( oldA4 );
} /* end GetMouse68kStub() */

#endif      /* 68k code */


/*
    @@@@@@@@@@@@@@@  Shared Code   @@@@@@@@@@@@@@@

    This code gets compiled into both 68k and powerpc object code.
    The 68k code gets called from 68k patches & code.
    The powerpc code gets called from powerpc patches & code.

    If these routines were very large, or called infrequently, we could
    just have a single version that is called by the "other" object code,
    but it's not worth the hassle & context switch.
*/
Handle Get1ResourceSys( OSType rezType, short rezID )
{
    THz         oldZone;
    Handle      h;

    oldZone = GetZone();
    SetZone( SystemZone() );
    h = Get1Resource( rezType, rezID );
```

```
   SetZone( oldZone );
   return h;
} /* end Get1ResourceSys() */

/*   Our custom GetMouse function. We do our screen stuff here because
   _GetMouse is allowed to move memory, and is called frequently.
*/

void MyGetMouse( Point *pt )
{
long   currentDepth;

   if ( gGlobalsPtr->gRequestFlag )/* Event is for us ? */
      {
      gGlobalsPtr->gRequestFlag = FALSE;
/* Clear flag or else get called indefinitely */
      currentDepth = Fetch_Depth();
      if ((currentDepth == BLACK_WHITE) && (currentDepth != gGlobalsPtr-
        ➥>gOldScreenDepth))
         Change_Depth(gGlobalsPtr->gOldScreenDepth);
      else
         Change_Depth(BLACK_WHITE);
      } /* end if */


   CallGetMouse( pt );            /* Hop to original GetMouse() */

} /* end ourGetMouse() */

#ifdef powerc

OSErr MyPostEventPPC( short eventNum, long eventMsg )
{
OSErr   result;

   if ( (eventNum == keyDown) || (eventNum == autoKey) )
      {
      if ( (lowMemKeyModifiers == KEY_COMBO) && (lowMemKeyStroke == T_KEYCODE) )
         {
         eventNum = nullEvent;   /* Supress the event */
         gGlobalsPtr->gRequestFlag = TRUE;
         } /* end if KEY_COMBO && T_KEYCODE */
      } /* end if */
```

```
   result = CallPostEvent(eventNum, eventMsg);
   return result;
} /* end MyPostEventPPC() */

#else      // 68K code

/*
   Note:
   returns the (possibly modified) event code

   Don't modify the local variables eventNum & eventMsg
      -- they're used by the stub routine and modifying
      -- locals here can have a global effect
*/
short MyPostEvent68k( short eventNum, long eventMsg )
{
short   newEventCode = eventNum;

   if ( (eventNum == keyDown) || (eventNum == autoKey) )
      {
      if ( (lowMemKeyModifiers == KEY_COMBO) && (lowMemKeyStroke == T_KEYCODE) )
         {
         newEventCode = nullEvent;   /* Supress the event */
         gGlobalsPtr->gRequestFlag = TRUE;
         } /* end if KEY_COMBO && T_KEYCODE */
      } /* end if */

   return newEventCode;
} /* end MyPostEvent68k() */

#endif

/*   Get the current screen depth. Also get the GDevice of main screen and its
   device number (to use the driver) */
long Fetch_Depth(void)
{
long      screenDepth;      /* Current bit depth of our screen */
GDHandle   thisGDevice;

   thisGDevice = GetMainDevice();         /* Get GDevice of main screen */
   gGlobalsPtr->gOurGDevice = thisGDevice;   /* Hang onto this gDevice's handle */
   screenDepth = (**thisGDevice).gdMode;   /* Get current video mode */
```

```
   gGlobalsPtr->gDevRefNum = (**thisGDevice).gdRefNum;
   /* Get screen's device ref. number */
   return screenDepth;

} /* end Fetch_Depth() */


/*
   Change screen depth. New screen depth is resource ID of
   a display mode the video hardware supports.
*/
void Change_Depth(long newDepth)
{
GrafPtr      oldPort;
Rect      ourGDRect;
RgnHandle   thisScreenBoundary;
GrafPtr      theBigPicture;
WindowPtr   theFrontWindow;

   HideCursor();            /* Hide pointer since it's depth will change */
   InitGDevice(gGlobalsPtr->gDevRefNum, newDepth, gGlobalsPtr->gOurGDevice);
/* At last we change the screen depth! */
   theFrontWindow = FrontWindow();
   ActivatePalette(theFrontWindow);
 /* Use active window's color palette */
   AllocCursor();            /* Draw cursor at new screen depth */
   ShowCursor();            /* Put it back on-screen */

/* The desktop's still a mess: redraw it */
   thisScreenBoundary = NewRgn();/* Get a region to hold this screen */
   if (!MemError())         /* Trouble? */
      {                  /* No */
      ourGDRect = (**gGlobalsPtr->gOurGDevice).gdRect;
      RectRgn(thisScreenBoundary, &ourGDRect);
/* Get boundary of gDevice */
      GetPort(&oldPort);      /* Save current port */
      GetWMgrPort(&theBigPicture);      /* Get Desktop's port */
      SetPort(theBigPicture);   /* Make it the current port */
      DrawMenuBar();
      PaintOne(NIL, thisScreenBoundary);
/* Paint the background */
      PaintBehind(LMGetWindowList(), thisScreenBoundary);
/* Now the other windows */
```

```
      SetPort(oldPort);
      DisposeRgn(thisScreenBoundary);
      } /* end if !MemError() */
   else
      SysBeep(30);


/* Couldn't make the region, complain */

} /* end Change_Depth() */
```

FlipDepth.µ.PPC.exp

```
gGlobalsPtr
Change_Depth
Fetch_Depth
MyPostEventPPC
MyGetMouse
Get1ResourceSys
```

FlipDepth.r

```
#include "SysTypes.r"
#include "Types.r"

#define EXTENSION_FREF      128
/* Resource IDs for file refs & icons */
#define BAD_LOAD_FREF      129
#define FLIPDEPTH_ICON      128
#define BAD_LOAD_ICON      129


resource 'BNDL' (128)
{
   'FLDP', 0,
   {
   'FREF', { 0, EXTENSION_FREF, 0, BAD_LOAD_FREF },
   'ICN#', { 0, FLIPDEPTH_ICON, 1, BAD_LOAD_ICON }
   }
};
```

```
resource 'FREF' (EXTENSION_FREF)
{
   'INIT',
   0,
   ""
};
resource 'FREF' (BAD_LOAD_FREF)
{
   'BADL',
   1,
   ""
};

/* Signature resource - all 'STR ' resources must be declared before this! */

type 'FLDP' as 'STR ';

resource 'FLDP' (0) {
    "FlipDepth 1.2"
};


data 'ICN#' (FLIPDEPTH_ICON) {
   $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
   $"6000 0006 4000 0002 8000 0001 83FF FFC1"
   $"83FF FFC1 8EAA AA81 8D55 5511 8EAA AA11"
   $"8D55 5411 8EAA A811 8D55 4011 8EAA 8011"
   $"8D55 0011 8EAA 0011 8D54 0011 8EA0 0011"
   $"8D40 0011 8E80 0011 8D00 0011 8000 0001"
   $"83FF FFC1 8000 0001 4E00 0002 6000 0006"
   $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
   $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
   $"7FFF FFFE 7FFF FFFE FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
   $"FFFF FFFF FFFF FFFF 7FFF FFFE 7FFF FFFE"
   $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
};

data 'ICN#' (BAD_LOAD_ICON) {
   $"0000 0000 0000 0000 0000 0000 1FFF FFF8"
```

```
    $"6800 0026 5C00 0072 8E00 00E1 87FF FFC1"
    $"83FF FFC1 8FEA AF91 8DF5 5F11 8EFA BE11"
    $"8D7D 7C11 8EBE F811 8D5F E011 8EAF C011"
    $"8D57 C011 8EAF E011 8D5E 7011 8EBC 3811"
    $"8D78 1C11 8EF0 0E11 8DE0 0711 81C0 0381"
    $"83BF FDC1 8700 00E1 4E00 0072 6400 0026"
    $"1FFF FFF8 0000 0000 0000 0000 0000 0000"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
};

data 'icl8' (FLIPDEPTH_ICON, purgeable) {
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 00FF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FF00 0000"
    $"00FF FF00 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 00FF FC00"
    $"00FF 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 FF00"
    $"FF00 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 002B 2BFF"
    $"FF00 0000 0000 FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF 0000 2B2B 2BFF"
    $"FF00 0000 0000 FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF 002B 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF00 FF00 FF00"
    $"FF00 FF00 FF00 FF00 FF2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 00FF 00FF 00FF"
    $"00FF 00FF 00FF 00FF 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF00 FF00 FF00"
    $"FF00 FF00 FF00 FF2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 00FF 00FF 00FF"
```

```
    $"00FF 00FF 00FF 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF00 FF00 FF00"
    $"FF00 FF2A FF2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 00FF 00FF 00FF"
    $"00FF 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF00 FF00 FF00"
    $"FF2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 00FF 00FF 00FF"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF00 FF00 FF2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 00FF 2AFF 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF00 FF2A 2A2A 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 00FF 2A2A 2A2A 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF FF00 FF2A 2A2A 2A2A 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 FFFF 00FF 2A2A 2A2A 2A2A 2A2A"
    $"2A2A 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 2B2B 2B2B 2BFF"
    $"FF00 0000 002B 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 2B2B 2B2B 2BFF"
    $"FF00 0000 2B2B 2B2B 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B 2B2B 2B2B 2B2B 2BFF"
    $"00FF 002B D8D8 D82B 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B 2B2B 2B2B 2B2B FF00"
    $"00FF FF2B 2B2B 2B2B 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B 2B2B 2B2B 2BFF FC00"
    $"0000 00FF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FF00 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
};

data 'icl8' (BAD_LOAD_ICON) {
    $"0000 0000 0000 0000 0000 0000 0000 0000"
```

```
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 00FF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF FFFF FF00 0000"
$"00FF FF00 D800 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 D800 00FF FC00"
$"00FF 00D8 D8D8 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 00D8 D8D8 0000 FF00"
$"FF00 0000 D8D8 D800 0000 0000 0000 0000"
$"0000 0000 0000 0000 D8D8 D800 002B 2BFF"
$"FF00 0000 00D8 D8D8 FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFD8 D8D8 0000 2B2B 2BFF"
$"FF00 0000 0000 D8D8 D8FF FFFF FFFF FFFF"
$"FFFF FFFF FFFF D8D8 D8FF 002B 2B2B 2BFF"
$"FF00 0000 FFFF FFD8 D8D8 FF00 FF00 FF00"
$"FF00 FF00 FFD8 D8D8 FF2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF D8D8 D8FF 00FF 00FF"
$"00FF 00FF D8D8 D8FF 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FFD8 D8D8 FF00 FF00"
$"FF00 FFD8 D8D8 FF2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF 00FF D8D8 D8FF 00FF"
$"00FF D8D8 D8FF 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FF00 FFD8 D8D8 FF00"
$"FFD8 D8D8 FF2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF 00FF 00FF D8D8 D8FF"
$"D8D8 D82A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FF00 FF00 FFD8 D8D8"
$"D8D8 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF 00FF 00FF 00FF D8D8"
$"D8D8 2A2A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FF00 FF00 FFD8 D8D8"
$"D8D8 D82A 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF 00FF 00FF D8D8 D82A"
$"2AD8 D8D8 2A2A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FF00 FFD8 D8D8 2A2A"
$"2A2A D8D8 D82A 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF 00FF 00FF D8D8 D82A 2A2A"
$"2A2A 2AD8 D8D8 2A2A 2A2A 0000 2B2B 2BFF"
$"FF00 0000 FFFF FF00 FFD8 D8D8 2A2A 2A2A"
$"2A2A 2A2A D8D8 D82A 2A2A 0000 2B2B 2BFF"
```

```
    $"FF00 0000 FFFF 00FF D8D8 D82A 2A2A 2A2A"
    $"2A2A 2A2A 2AD8 D8D8 2A2A 0000 2B2B 2BFF"
    $"FF00 0000 0000 00D8 D8D8 0000 0000 0000"
    $"0000 0000 0000 D8D8 D800 2B2B 2B2B 2BFF"
    $"FF00 0000 002B D8D8 D800 0000 0000 0000"
    $"0000 0000 0000 00D8 D8D8 2B2B 2B2B 2BFF"
    $"FF00 0000 2BD8 D8D8 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B D8D8 D82B 2B2B 2BFF"
    $"00FF 002B D8D8 D82B 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B 2BD8 D8D8 2B2B FF00"
    $"00FF FF2B 2BD8 2B2B 2B2B 2B2B 2B2B 2B2B"
    $"2B2B 2B2B 2B2B 2B2B 2B2B D82B 2BFF FC00"
    $"0000 00FF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FF00 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
};

data 'cicn' (FLIPDEPTH_ICON) {
    $"0000 0000 8010 0000 0000 0020 0020 0000"
    $"0000 0000 0000 0048 0000 0048 0000 0000"
    $"0004 0001 0004 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0004 0000 0000 0020 0020"
    $"0000 0000 0004 0000 0000 0020 0020 0000"
    $"0000 0000 0000 0000 0000 0000 0000 1FFF"
    $"FFF8 7FFF FFFE 7FFF FFFE FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF 7FFF FFFE 7FFF"
    $"FFFE 1FFF FFF8 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 1FFF"
    $"FFF8 6000 0006 4000 0002 8000 0001 83FF"
    $"FFC1 83FF FFC1 8EAA AA81 8D55 5511 8EAA"
    $"AA11 8D55 5411 8EAA A811 8D55 4011 8EAA"
    $"8011 8D55 0011 8EAA 0011 8D54 0011 8EA0"
    $"0011 8D40 0011 8E80 0011 8D00 0011 8000"
    $"0001 83FF FFC1 8000 0001 4E00 0002 6000"
```

```
    $"0006 1FFF FFF8 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0005 0000 FFFF FFFF"
    $"FFFF 0001 DDDD 0000 0000 0002 CCCC CCCC"
    $"FFFF 0003 CCCC CCCC CCCC 0004 4444 4444"
    $"4444 000F 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 000F FFFF FFFF"
    $"FFFF FFFF FFFF FFFF F000 0FF0 0000 0000"
    $"0000 0000 0000 0000 0F40 0F00 0000 0000"
    $"0000 0000 0000 0000 00F0 F000 0000 0000"
    $"0000 0000 0000 0000 033F F000 00FF FFFF"
    $"FFFF FFFF FFFF FF00 333F F000 00FF FFFF"
    $"FFFF FFFF FFFF FF03 333F F000 FFF0 F0F0"
    $"F0F0 F0F0 F0F0 F200 333F F000 FF0F 0F0F"
    $"0F0F 0F0F 0F0F 2200 333F F000 FFF0 F0F0"
    $"F0F0 F0F0 F0F2 2200 333F F000 FF0F 0F0F"
    $"0F0F 0F0F 0F22 2200 333F F000 FFF0 F0F0"
    $"F0F0 F0F2 F222 2200 333F F000 FF0F 0F0F"
    $"0F0F 0F22 2222 2200 333F F000 FFF0 F0F0"
    $"F0F0 F222 2222 2200 333F F000 FF0F 0F0F"
    $"0F0F 2222 2222 2200 333F F000 FFF0 F0F0"
    $"F0F2 2222 2222 2200 333F F000 FF0F 0F0F"
    $"2F22 2222 2222 2200 333F F000 FFF0 F0F2"
    $"2222 2222 2222 2200 333F F000 FF0F 0F22"
    $"2222 2222 2222 2200 333F F000 FFF0 F222"
    $"2222 2222 2222 2200 333F F000 FF0F 2222"
    $"2222 2222 2222 2200 333F F000 0000 0000"
    $"0000 0000 0000 0033 333F F000 0300 0000"
    $"0000 0000 0000 0033 333F F000 3333 3333"
    $"3333 3333 3333 3333 333F 0F03 1113 3333"
    $"3333 3333 3333 3333 33F0 0FF3 3333 3333"
    $"3333 3333 3333 3333 3F40 000F FFFF FFFF"
    $"FFFF FFFF FFFF FFFF F000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000"
};

data 'cicn' (BAD_LOAD_ICON) {
    $"0000 0000 8010 0000 0000 0020 0020 0000"
    $"0000 0000 0000 0048 0000 0048 0000 0000"
    $"0004 0001 0004 0000 0000 0000 0000 0000"
```

```
$"0000 0000 0000 0004 0000 0000 0020 0020"
$"0000 0000 0004 0000 0000 0020 0020 0000"
$"0000 0000 0000 0000 0000 0000 0000 1FFF"
$"FFF8 7FFF FFFE 7FFF FFFE FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
$"FFFF FFFF FFFF FFFF FFFF 7FFF FFFE 7FFF"
$"FFFE 1FFF FFF8 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 1FFF"
$"FFF8 6800 0026 5C00 0072 8E00 00E1 87FF"
$"FFC1 83FF FFC1 8FEA AF91 8DF5 5F11 8EFA"
$"BE11 8D7D 7C11 8EBE F811 8D5F E011 8EAF"
$"C011 8D57 C011 8EAF E011 8D5E 7011 8EBC"
$"3811 8D78 1C11 8EF0 0E11 8DE0 0711 81C0"
$"0381 83BF FDC1 8700 00E1 4E00 0072 6400"
$"0026 1FFF FFF8 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0005 0000 FFFF FFFF"
$"FFFF 0001 DDDD 0000 0000 0002 CCCC CCCC"
$"FFFF 0003 CCCC CCCC CCCC 0004 4444 4444"
$"4444 000F 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 0000 0000 0000"
$"0000 0000 0000 0000 0000 000F FFFF FFFF"
$"FFFF FFFF FFFF FFFF F000 0FF0 1000 0000"
$"0000 0000 0000 0010 0F40 0F01 1100 0000"
$"0000 0000 0000 0111 00F0 F000 1110 0000"
$"0000 0000 0000 1110 033F F000 0111 FFFF"
$"FFFF FFFF FFF1 1100 333F F000 0011 1FFF"
$"FFFF FFFF FF11 1F03 333F F000 FFF1 11F0"
$"F0F0 F0F0 F111 F200 333F F000 FF0F 111F"
$"0F0F 0F0F 111F 2200 333F F000 FFF0 F111"
$"F0F0 F0F1 11F2 2200 333F F000 FF0F 0F11"
$"1F0F 0F11 1F22 2200 333F F000 FFF0 F0F1"
$"11F0 F111 F222 2200 333F F000 FF0F 0F0F"
$"111F 1112 2222 2200 333F F000 FFF0 F0F0"
$"F111 1122 2222 2200 333F F000 FF0F 0F0F"
$"0F11 1122 2222 2200 333F F000 FFF0 F0F0"
$"F111 1112 2222 2200 333F F000 FF0F 0F0F"
$"1112 2111 2222 2200 333F F000 FFF0 F0F1"
$"1122 2211 1222 2200 333F F000 FF0F 0F11"
$"1222 2221 1122 2200 333F F000 FFF0 F111"
```

```
    $"2222 2222 1112 2200 333F F000 FF0F 1112"
    $"2222 2222 2111 2200 333F F000 0001 1100"
    $"0000 0000 0011 1033 333F F000 0311 1000"
    $"0000 0000 0001 1133 333F F000 3111 3333"
    $"3333 3333 3333 1113 333F 0F03 1113 3333"
    $"3333 3333 3333 3111 33F0 0FF3 3133 3333"
    $"3333 3333 3333 3313 3F40 000F FFFF FFFF"
    $"FFFF FFFF FFFF FFFF F000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0000 0000 0000 0000"
};
```

Klepto.c

```
#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Fonts.h>
#include <Memory.h>
#include <ToolUtils.h>
#include <StandardFile.h>
#include <Errors.h>
#include <Resources.h>

/* Various constants */
#define NIL             0L
#define FALSE          false
#define TRUE           true
#define DEFAULT_VOL      0
#define ONE_FILE_TYPE    1
#define POWER_PC_FRAG    'PPC '                  /* Resource type */
#define FRAG_ID          300                     /* Resource ID */

void Move_Fork(short input);
void main(void);

void Move_Fork(short input)
{
OSErr          fInputErr;
long          codeFragSize;
Handle          fragBuff;
```

```
   fInputErr = GetEOF(input, &codeFragSize);      /* Get file length */
   if ((fragBuff = NewHandle(codeFragSize)) != NIL)/* Enough data buffer memory? */
      {
      if (!(fInputErr = FSRead(input, &codeFragSize, *fragBuff)))
/* Read in fragment */
         {
         AddResource(fragBuff, POWER_PC_FRAG, FRAG_ID, NIL);
/* Treat buffer as a resource */
         if (!ResError())                  /* Trouble? */
            {
            WriteResource(fragBuff);         /* Write frag to resource fork */
            if (ResError())
               SysBeep(30);
            } /* end if !ResError */
         } /* !fInputErr */
      } /* end if != NIL */
   UpdateResFile(CurResFile());               /* Update resource map */
   ReleaseResource(fragBuff);                  /* Free the memory */
} /* end Move_Fork() */

void main(void)
{
unsigned char      fileName[21] = {"\pFlipDepth.μ.PPC.rsrc"};
OSType            fileCreator = {'RSED'};
/* File type and creator for our output file */
OSType            fileType = {'rsrc'};
OSErr             fileError;
short             inFileRefNum, outFileRefNum;
StandardFileReply   inputReply, outputReply;
short             oldVol;
SFTypeList         shlbType = {'shlb'};      /* File type for shared libraries */
CursHandle         theCursor;      /* Current pointer icon */


/* Lunge after all the memory we can get */
   MaxApplZone();

/* Make sure we've got some master pointers */
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();
```

```
/* Initialize managers */
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(NIL);

/* Open the input file */
   StandardGetFile(NIL, ONE_FILE_TYPE, shlbType, &inputReply);
   if (inputReply.sfGood)
      {
      GetVol (NIL, &oldVol);   /* Save current volume */
         if ((fileError = FSpOpenDF (&inputReply.sfFile, fsCurPerm,
            ➥&inFileRefNum)) != noErr)
         {
         SysBeep(30);
         return;
         } /* end if error */

/* Open the output file */
      StandardPutFile ("\pSave code fragment in:", fileName, &outputReply);
      if (outputReply.sfGood)
         {
         SetVol(NIL, outputReply.sfFile.vRefNum);
/* Make the destination volume current */
         fileError = FSpCreate(&outputReply.sfFile, fileCreator, fileType,
            ➥smSystemScript);
         switch(fileError) /* Process result from File Manager */
            {
            case noErr:
            break;
            case dupFNErr /* File already exists, wipe it out */
               if ((fileError = FSpDelete(&outputReply.sfFile)) == noErr)
                  {
                  if ((fileError = FSpCreate(&outputReply.sfFile, fileCreator,
                                    fileType, smSystemScript)) != noErr)
                     {
                     SysBeep(30);
                     FSClose (inFileRefNum);
                     SetVol(NIL, oldVol);
```

```
                    return;
                    } /* end if != noErr */
                  } /* end == noErr */
               else
                  {
                  SysBeep(30);
                  FSClose (inFileRefNum);
                  SetVol(NIL, oldVol);
                  return;
                  } /* end else */
            break;   /* end case dupFNErr */
            default:  /* Unknown error, try to abort cleanly */
               SysBeep(30);
               FSClose (inFileRefNum);
/* Close the input file */
               SetVol(NIL, oldVol);
/* Restore original volume */
               return;
            } /* end switch */

/* Open file's data fork.*/
/* We do this only to get a file ref number */
         if (!(FSpOpenDF (&outputReply.sfFile, fsCurPerm, &outFileRefNum)))
/* Open data fork */
            {
/* MUST create resource map in resource fork or no resource writing occurs */
            FSpCreateResFile (&outputReply.sfFile, fileCreator, fileType,
              ➥smSystemScript);
            if (!ResError())
               {
               FSpOpenResFile (&outputReply.sfFile, fsCurPerm);
/* Open resource fork */
               if (!ResError())
                  {
                  theCursor = GetCursor(watchCursor);
 /* Change the cursor */
                  SetCursor(&**theCursor);
                  Move_Fork (inFileRefNum);
                  FSClose (outFileRefNum);
                  SetCursor(&qd.arrow); /* Restore the cursor */
                  } /* end if !ResError */
               } /* end if !ResError */
            FlushVol (NIL, outputReply.sfFile.vRefNum);
```

```
            } /* end if !FSpOpenDF */
          } /* end if outputReply.sfGood */
        FSClose (inFileRefNum);
        SetVol(NIL, oldVol);   /* Restore current volume */
    } /* end if inputReply.sfGood */

} /* end main() */
```

ShowInitIcon.h

```
#ifndef __ShowInitIcon__
#define __ShowInitIcon__

#include <Types.h>

// Usage: pass the ID of your icon family (ICN#/icl4/icl8) to have it drawn in the
// right spot.
// If 'advance' is true, the next INIT icon will be drawn to the right of your
// icon. If it is false, the next INIT icon will overwrite
// yours. You can use it to create animation effects by calling ShowInitIcon
// several times with 'advance' set to false.

#ifdef __cplusplus
extern "C" {
#endif

pascal void ShowInitIcon (short iconFamilyID, Boolean advance);

#ifdef __cplusplus
}
#endif

#endif /* __ShowInitIcon__ */
```

ShowInitIcon.c

```
// ShowInitIcon - version 1.0.1, May 30th, 1995
// This code is intended to let INIT writers easily display an icon at startup time.
// View in Geneva 9pt, 4-space tabs

// Written by: Peter N Lewis <peter@mail.peter.com.au>, Jim Walker
// <JWWalker@aol.com>
```

```
// and François Pottier <pottier@dmi.ens.fr>, with thanks to previous ShowINIT
// authors.
// Send comments and bug reports to François Pottier.

// This version features:
// - Short and readable code.
// - Correctly wraps around when more than one row of icons has been displayed.
// - works with System 6
// - Built with Universal Headers & CodeWarrior. Should work with other headers/
// compilers.

#define USESROUTINEDESCRIPTORS GENERATINGCFM

#include <Memory.h>
#include <Resources.h>
#include <Icons.h>
#include <OSUtils.h>
#include <LowMem.h>
#include "ShowInitIcon.h"

#define SYSTEM_7      0x0700


// You should set SystemSixOrLater in your headers to avoid including glue for
SysEnvirons.

// ------------------------------------------------------------------------
// Set this flag to 1 if you want to compile this file into a stand-alone resource
// (see note below).
// Set it to 0 if you want to include this source file into your INIT project.

#define STAND_ALONE_RESOURCE   1

#if STAND_ALONE_RESOURCE

#define ShowInitIcon main
// For the linker, which expects a "main" symbol

enum
{
   kShowInitIconInfo = kPascalStackBased
               ¦ STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
               ¦ STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(Boolean)))
};
```

```
ProcInfoType __procinfo   = kShowInitIconInfo;
 // Resource header info, is 0x180

#endif


// ------------------------------------------------------------------------
// The ShowINIT mechanism works by having each INIT read/write data from these
// globals.
// The MPW C compiler doesn't accept variables declared at an absolute address, so
// I use these macros instead.
// Only one macro is defined per variable; there is no need to define a Set and a
// Get accessor like in <LowMem.h>.

#define   LMVCoord         (*(short *) (LMGetCurApName() + 32 - 6))
//#define   LMVCoord          (* (short*) 0x92A)
#define   LMVCheckSum      (* (short *) (LMGetCurApName() + 32 - 8))
//#define   LMVCheckSum       (* (short*) 0x928)
#define   LMHCoord         (* (short *) (LMGetCurApName() + 32 - 4))
//#define   LMHCoord          (* (short*) 0x92C)
#define   LMHCheckSum      (* (short *) (LMGetCurApName() + 32 - 2))
//#define   LMHCheckSum       (* (short*) 0x92E)
// ------------------------------------------------------------------------
// Prototypes for the subroutines. The main routine comes first; this is necessary
// to make THINK C's "Custom Header" option work.

static unsigned short CheckSum (unsigned short x);
static void ComputeIconRect (Rect* iconRect, Rect* screenBounds);
static void AdvanceIconPosition (Rect* iconRect);
static void DrawBWIcon (short iconID, Rect *iconRect);

// ------------------------------------------------------------------------
// Main routine.

typedef struct {
   QDGlobals         qd;  // Storage for the QuickDraw globals
   long            qdGlobalsPtr;
  // A5 points to this place; it will contain a pointer to qd
  } QDStorage;
```

```
pascal void ShowInitIcon (short iconFamilyID, Boolean advance)
{
   long            oldA5;   // Original value of register A5
   QDStorage         qds;  // Fake QD globals
   CGrafPort         colorPort;
   GrafPort         bwPort;
   Rect            destRect;
   SysEnvRec      environment;   // Machine configuration.

   oldA5 = SetA5((long) &qds.qdGlobalsPtr);
// Tell A5 to point to the end of the fake QD Globals
   InitGraf(&qds.qd.thePort); // Initialize the fake QD Globals

   SysEnvirons(curSysEnvVers, &environment);
// Find out what kind of machine this is

   ComputeIconRect(&destRect, &qds.qd.screenBits.bounds);
// Compute where the icon should be drawn

   if (environment.systemVersion >= SYSTEM_7 && environment.hasColorQD) {
      OpenCPort(&colorPort);
      PlotIconID(&destRect, atNone, ttNone, iconFamilyID);
      CloseCPort(&colorPort);
   }
   else {
      OpenPort(&bwPort);
      DrawBWIcon(iconFamilyID, &destRect);
      ClosePort(&bwPort);
   }

   if (advance)
      AdvanceIconPosition (&destRect);

   SetA5(oldA5);  // Restore A5 to its previous value
}

// ---------------------------------------------------------------------------
// A checksum is used to make sure that the data in there was left by another
// ShowINIT-aware INIT.

static unsigned short CheckSum (unsigned short x)
{
   return ((x << 1) | (x >> 15)) ^ 0x1021;
```

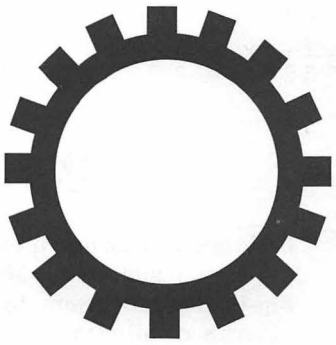
```
}

// ----------------------------------------------------------------------------
// ComputeIconRect computes where the icon should be displayed.

static void ComputeIconRect (Rect* iconRect, Rect* screenBounds)
{
   if (CheckSum(LMHCoord) != LMHCheckSum)
// If we are first, we need to initialize the shared data.
      LMHCoord = 8;
   if (CheckSum(LMVCoord) != LMVCheckSum)
      LMVCoord = screenBounds->bottom - 40;

   if (LMHCoord + 34 > screenBounds->right) {
  // Check whether we must wrap
      iconRect->left = 8;
      iconRect->top = LMVCoord - 40;
   }
   else {
      iconRect->left = LMHCoord;
      iconRect->top = LMVCoord;
   }
   iconRect->right = iconRect->left + 32;
   iconRect->bottom = iconRect->top + 32;
}

// AdvanceIconPosition updates the shared global variables so that the next
// extension will draw its icon beside ours.

static void AdvanceIconPosition (Rect* iconRect)
{
   LMHCoord = iconRect->left + 40;
// Update the shared data
   LMVCoord = iconRect->top;
   LMHCheckSum = CheckSum(LMHCoord);
   LMVCheckSum = CheckSum(LMVCoord);
}

// DrawBWIcon draws the 'ICN#' member of the icon family. It works under System 6.

static void DrawBWIcon (short iconID, Rect *iconRect)
{
```

```
    Handle      icon;
    BitMap      source, destination;
    GrafPtr      port;

    icon = Get1Resource('ICN#', iconID);
    if (icon != NULL) {
       HLock(icon);
       // Prepare the source and destination bitmaps.
       source.baseAddr = *icon + 128; // Mask address.
       source.rowBytes = 4;
       SetRect(&source.bounds, 0, 0, 32, 32);
       GetPort(&port);
       destination = port->portBits;
             // Transfer the mask.
       CopyBits(&source, &destination, &source.bounds, iconRect, srcBic, nil);
                                          // Then the icon.
       source.baseAddr = *icon;
       CopyBits(&source, &destination, &source.bounds, iconRect, srcOr, nil);
    }
}

// ----------------------------------------------------------------------------
// Notes

// Checking for PlotIconID:
// We (PNL) now check for system 7 and colour QD, and use colour graf ports and
// PlotIconID only if both are true
// Otherwise we use B&W grafport and draw using PlotBWIcon.
```

FatCodeResource.r

```
#include "MixedMode.r"
// Enter the ProcInfo type as a hexadecimal value,
// as either $01, or 0x01. Use the CodeWarrior
// disassembler to determine this value

// Use resource type 'fdes' for a fat resource (PPC and 68K Macs running Mixed Mode)
```

```
// Use resource type 'sdes' for s safe fat resource (runs on all Macs)

resource 'sdes' (128, sysheap, locked) {
   $180,                                  // 68K ProcInfo
   $180,                                  // PowerPC ProcInfo
   $$Resource("68KCodeResource", 'MWCW', 128),
// Specify filename, type, and ID of resource
    //   containing 68K code
      $$Resource("PPCCodeResource", 'MWCW', 128),
      // Specify filename, type, and ID of resource
    //   containing a PEF container
};
```



Index

Symbols

A

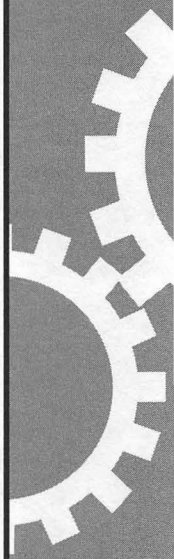
B

C

D

E

F



G

H

I

J-L

M

N

O-P

Q-R

S

T

U-V

W-Z

Become a CodeWarrior now!

Order the commercial version of Metrowerks CodeWarrior!

see other side for ordering information

Metrowerks CodeWarrior delivers three times a year. When you buy CodeWarrior and register with Metrowerks, you will receive free updates throughout the year.

Bronze (For 68K Macintosh)	**$149.00**
Gold (For Power & 68K Macintosh, Win32, Magic Cap, Be, Java)	**$399.00**

Metrowerks CodeWarrior. The world's best-selling Macintosh development tools.

Metrowerks is continually adding new features and products. Check our website for the latest products, prices and Geekware.

Metrowerks CodeWarrior Order Form

metrowerks®

Gold @US$399 ea. X ___ = ______
Bronze @US$149 ea. X ___ = ______
Subtotal ______
Plus sales tax & shipping ______
(as may apply)
Total ______

Method of Payment
❑ VISA
❑ Mastercard

Exp. Date (M/Y)

Credit Card Number

First & Last name

Street

City/State/Prov. Zip/Postal Code

Email address

Phone number

Signature Date Ordered

Fax to: (512) 305-0440 or call (800) 377-5416

or Mail to: **Metrowerks Corp**
Dept 334
P.O. Box 9700
Austin, TX 78766-9700

For Sales info:
WWW: http://www.metrowerks.com
Voice: (512) 305-0400
Fax: (512) 305-0440
Email: sales@metrowerks.com

Prices and product availability may change without notice - check our website for the latest information.

Software License

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE.

BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system, and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Software"), the related documentation, and fonts are licensed to you by Metrowerks. You own the disk on which the Software and fonts are recorded, but Metrowerks and/or Metrowerks' Licensors retain title to the Software, related documentation, and fonts. This License allows you to use the Software and fonts on a single Apple computer and make one copy of the Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the Metrowerks copyright notice and any other proprietary legends that were on the original copy of the Software and fonts. You may also transfer all your license rights in the Software and fonts, the backup copy of the Software and fonts, the related documentation, and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets, and other proprietary material. In order to protect them, and except as permitted by applicable legislation, you may not decompile, reverse engineer, disassemble, or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease,

loan, distribute, or create derivative works based upon the Software in whole or in part. You may not electronically transmit the Software from one computer to another or over a network.

3. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Software, related documentation and fonts, and all copies thereof. This License will terminate immediately without notice from Metrowerks if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and fonts, and all copies thereof.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Metrowerks, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Software has been rightfully obtained by you outside of the United States, you agree that you will not re-export the Software nor any other technical data received from Metrowerks, nor the direct product thereof, except as permitted by the laws and regulations of the United States and the laws and regulations of the jurisdiction in which you obtained the Software.

5. Government End Users. If you are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Software and fonts are supplied to the Department of Defense (DoD), the Software and fonts are classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, its documentation and fonts as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and

(ii) if the Software and fonts are supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Software, its documentation and fonts will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. Limited Warranty on Media. Metrowerks warrants the diskettes and/or compact disc on which the Software and fonts are recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Metrowerks' entire liability and your exclusive remedy will be replacement of the diskettes and/

or compact disc not meeting Metrowerks' limited warranty and which is returned to Metrowerks or a Metrowerks authorized representative with a copy of the receipt. Metrowerks will have no responsibility to replace a disk/disc damaged by accident, abuse, or misapplication. ANY IMPLIED WARRANTIES ON THE DISKETTES AND/OR COMPACT DISC, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY BY JURISDICTION.

7. Disclaimer of Warranty on Apple Software. You expressly acknowledge and agree that use of the Software and fonts is at your sole risk. The Software, related documentation and fonts are provided "AS IS" and without warranty of any kind and Metrowerks and Metrowerks' Licensor(s) (for the purposes of provisions 7 and 8, Metrowerks and Metrowerks' Licensor(s) shall be collectively referred to as "Metrowerks") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. METROWERKS DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, METROWERKS DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

8. Limitation of Liability. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL METROWERKS BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF METROWERKS OR A METROWERKS AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF

THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

In no event shall Metrowerks' total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Software and fonts.

9. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

10. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Metrowerks.

METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) MAKE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. METROWERKS DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

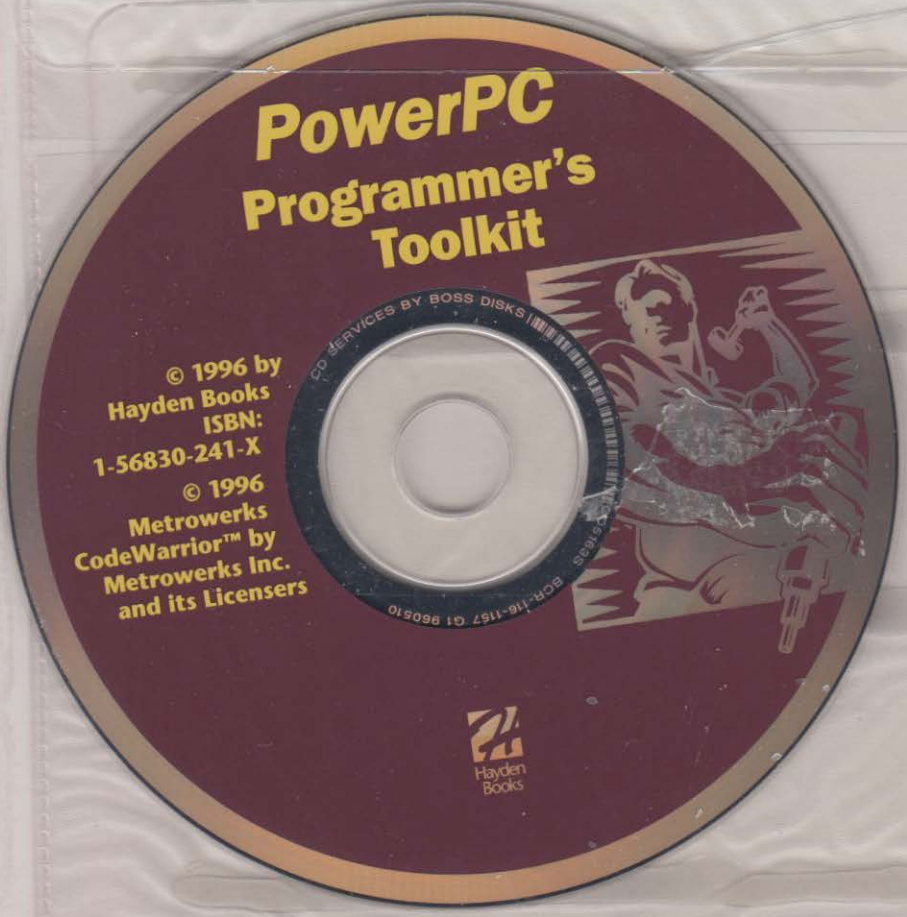
IN NO EVENT WILL METROWERKS AND METROWERKS' LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY METROWERKS) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF METROWERKS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Metrowerks liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to $1.

What You'll Find on the CD

The *PowerPC Programmer's Toolkit* CD contains Metrowerks CodeWarrior 8.0 Lite as well as all the sample code discussed in this book. This version of Metrowerks CodeWarrior is limited in that it can be used only with the code provided on the CD. Certain commands (such as New Project and Add File...) have been disabled. But, this version retains the functionality of CodeWarrior except for those functions. So, you can use almost all of CodeWarrior's features to learn to program your PowerMac.

Metrowerks cannot provide technical support for this limited version of CodeWarrior bundled with this book. If you have trouble, call Hayden Books at 1-800-858-7674 or email us at `hayden@hayden.com`.

Using the CD is simple; just pop it into your drive and dive in!


PowerPC
Programmer's
Toolkit
© 1996 by
Hayden Books
ISBN:
1-56830-241-X
© 1996
Metrowerks
CodeWarrior™ by
Metrowerks Inc.
and its Licensers
CD SERVICES BY BOSS DISKS
Hayden
Books

Everything You Need to Program for the ***PowerPC***™

PowerPC Programmer's Toolkit is your guide to the newest information and programming techniques for the next generation of computer processors, the PowerPC. This in-depth reference includes information on the entire PowerPC processor family, including the 601, 602, 603, 603e, 604, and the 604e.

Together with the special version of Metrowerks CodeWarrior™ Lite, the *PowerPC Programmer's Toolkit* will show you how to push the PowerPC to its limits. Contains everything you need to program for the PowerPC chip.

Tom Thompson is a Senior Technical Editor for *BYTE* magazine. He is an assistant Apple developer and has extensive programming experience. He has been involved with the Macintosh since its inception and is widely recognized as both a programmer and a writer.

CD-ROM includes a special version of Metrowerks CodeWarrior™ Lite and sample code from the book.

Discover how to...

- Write PowerPC applications in native code for blazing speed
- Utilize PowerPC code fragment TOC
- Understand and use UPPs (Univers Procedure Pointers
- Work with structur alignments and siz data structures
- Write, use, and control shared libraries

Approved for technical accuracy by Jim Trudeau, Document Manager at Metrowerks

$45.00 USA / $61.95 CAN
£41.50 Net UK (inc of VAT)

System 7.0 or higher
2 MB RAM
7 MB hard disk space

Category: Macintosh Programming
User Level: Intermediate – Advanced
Covers: PowerPC

Hayden Books

0 52059 02411 7

ISBN 1-56830-241-X

9 781568 302416